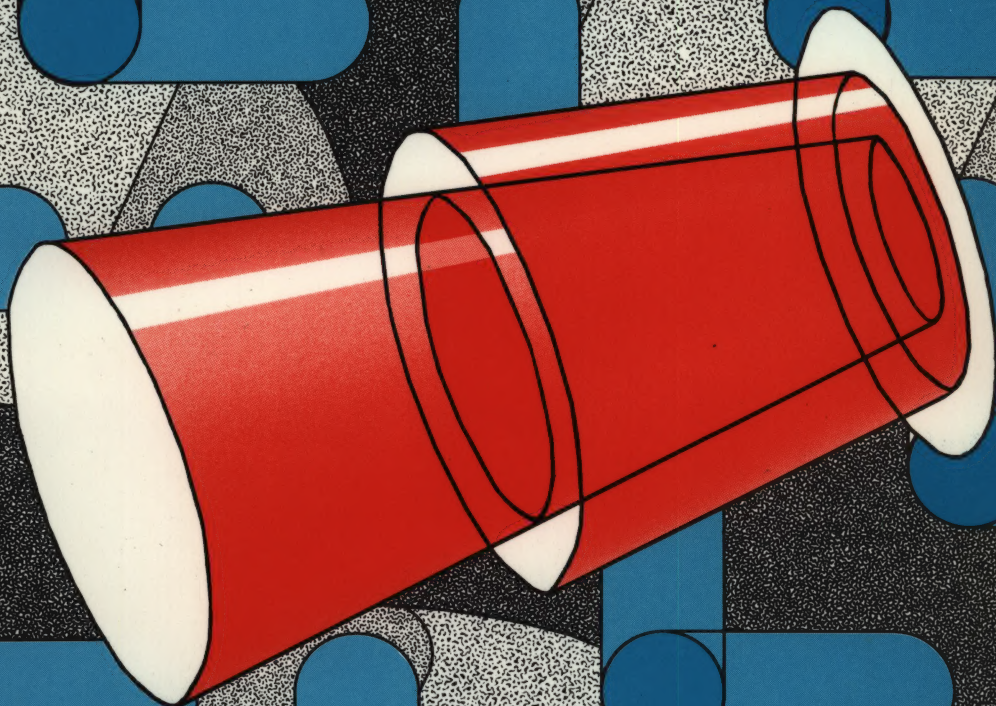
Inleiding in de berekenbaarheids-theorie

Dr. V.J. Rayward-Smith

ACADEMIC SERVICE



INLEIDING IN DE BEREKENBAARHEIDSTHEORIE

Serie onder redactie van **ir. J.J. van Amstel:**

Database, een inleiding - Date
Inleiding systeemanalyse en systeemontwerp - Davis
Software engineering - Sommerville
Problemen oplossen met de computer - Dromey
Compilerbouw - Wirth
Theorie en praktijk van besturingssystemen - Petersen & Silberschatz
Inleiding in de theorie van formele talen - Rayward-Smith
Inleiding in de berekenbaarheidstheorie - Rayward-Smith

Serie in samenwerking met Addison-Wesley:

Bestuurlijke informatieverzorging - van Swigchem

Inleiding in de berekenbaarheids-theorie

Dr. V.J. Rayward-Smith

Vertaling van: *A first course in computability 1e*
uitgegeven door Blackwell Scientific Publications Ltd
© 1985

Vertaling: Drs. M.M. Stefanski

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Rayward-Smith, V.J.

Inleiding in de berekenbaarheidstheorie / V.J. Rayward-Smith ; [vert. uit het Engels door M.M. Stefanski]. - Schoonhoven : Academic Service. - Ill.
Vert. van: A first course in computability. - Oxford : Blackwell Scientific, 1985. - (Computer science book).
ISBN 90-6233-243-9
SISO 520.6 SVS 8.12.3 UDC 519.68 NUGI 852
Trefw.: algoritmen / berekenbaarheidstheorie.

Uitgegeven door: Academic Service
Postbus 81
2870 AB Schoonhoven
Zetwerk: ATS, Lopik
Omslag: BSE Advertising, Alphen a/d Rijn
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam

ISBN 90 6233 243 9
NUGI 852

Inhoud

Voorwoord

Een groot deel van de informatica richt zich op het ontwerpen, analyseren en efficiënt uitwerken van algoritmen voor tal van taken. Dit boek tracht de meest fundamentele vragen die men zich bij dit soort activiteiten kan stellen te beantwoorden. Dat wil zeggen vragen als 'Waar liggen de grenzen van de mogelijkheden van computers?' en 'Bestaat er een algoritme voor het oplossen van een gegeven probleem?' en, zo ja 'Bestaat er ook een efficiënt algoritme?'. Deze vragen zijn zo uiterst belangrijk dat zij nauwgezet en gedetailleerd moeten worden beantwoord.

In dit boek wordt die nauwgezetheid bereikt door gebruik te maken van het beproefde instrumentarium van de wiskundige redenering. Misschien vinden sommige lezers dat afschrikwekkend, maar in feite is er geen alternatief. Het boek is zo opgezet dat elke informatica-student in zijn tweede of derde studiejaar de inhoud moet kunnen verwerken. Zelfs een eerstejaars student kan aanzienlijk profijt trekken uit het bestuderen van de tekst. Het begrip oplosbaarheid is een van de basisbegrippen in de informatica en de ontwikkeling van het begrip onoplosbaarheid kan leiden tot verrassende inzichten. Nadat een maat voor de efficiëntie van algoritme is ingevoerd, is het onderverdelen van oplosbare problemen in handelbare en onhandelbare problemen een vanzelfsprekende volgende stap. Het begrip onhandelbaarheid, ('intractability') wordt pas sinds vijftien jaar gehanteerd, maar moeilijk in deze theorie niet.

Theoretische informatici gaan er altijd van uit dat iedereen weet wat een Tuwingmachine is. Deze automaten bieden ons een eenvoudig model voor berekenbaarheid, waarmee fundamentele resultaten kunnen worden afgeleid. Juist omdat zij zo vaak worden gebruikt en algemeen aanvaard zijn als standaardmodel in de berekenbaarheidstheorie besteden wij er ook in dit boek aandacht aan. Natuurlijk bestaat er veel

den wij er ook in dit boek aandacht aan. Natuurlijk bestaan er veel andere modellen en soms hadden wij daarmee de presentatie kunnen vereenvoudigen, maar wij meenden de traditie ten aanzien van dit onderwerp te moeten respecteren. Zolang de meeste resultaten in publicaties nog worden gepresenteerd met behulp van Turingmachines is het van belang dat de informaticus van hun betekenis op de hoogte is.

Berekenbaarheidstheorie wordt vaak aan wiskundigen onderwezen en terecht! Het functiebegrip staat immers centraal in de wiskunde en het volledig doorzien daarvan zou een eerste vereiste moeten zijn voor het behalen van een universitaire graad in de mathematica. Helaas moet worden geconstateerd dat vele wiskunde faculteiten de berekenbaarheidstheorie degraderen tot een keuzevak in het laatste studiejaar. Wij hopen dat dit boek universitaire docenten ervan kan overtuigen dat het materiaal met succes in een vroeger stadium van de studie kan worden onderwezen.

Inleiding in de berekenbaarheidstheorie is één van drie boeken uit dezelfde serie die tezamen een theoretische grondslag vormen voor het informatica-onderwijs in de prekandidaatsfase. De andere twee boeken zijn *Inleiding in de Theorie van de Formele Talen* door V.J. Rayward-Smith en *Inleiding in de Formele Logica met Toepassingen in de Informatica* door R.D. Dowsing, V.J. Rayward-Smith en C.D. Walter.

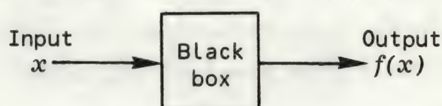
Tenslotte wil ik mijn collegae aan de universiteit van East Anglia, Norwich bedanken voor hun steun tijdens de voorbereiding van dit boek. In het bijzonder dank ik Dr. G.D. Smith voor zijn opmerkingen naar aanleiding van eerdere versies van de tekst. Ms. Carol Bracken dank ik voor haar typewerk; zonder haar hulp was dit boek een onleesbare stapel aantekeningen in potlood gebleven. Mr. Hugh Prior eveneens van de Universiteit van East Anglia ben ik dankbaar voor het ontwikkelen van het Pascal programma voor de Turing Simulator dat in de Appendix is opgenomen.

V.J. Rayward-Smith

Inleiding

Een naïeve computergebruiker kan dit apparaat zien als een 'black box' waarin hij of zij gegevens invoert en die resultaten oplevert. Van programmeren hoeft deze gebruiker niets te begrijpen; het programma is door iemand anders geschreven. De computer accepteert input, (bijvoorbeeld een rij symbolen x) en berekent output. De output is opnieuw een rij symbolen of een string en hangt in de meeste gevallen op de een of andere manier af van x. We kunnen de output dus beschouwen als een functie $f(x)$ van x. Hoe die functie en precies uitziet hangt af van het gebruikte programma; verschillende programma's berekenen meestal verschillende functies. (Zie Figuur 0.1.)

In dit boek wordt een model ontwikkeld voor de 'black box' en de programma's die erin kunnen zitten. Met behulp van dit model kunnen we alle mogelijke programma's en de functies die ze berekenen, bestuderen. Ons model moet liefst eenvoudig zijn, zodat we het gemakkelijk kunnen analyseren, maar het moet toch zo krachtig zijn dat we er alle functies mee kunnen berekenen die we op grond van onze ervaring berekenbaar achten. We noemen dergelijke functies 'effectief berekenbaar'. Het is moeilijk om het begrip 'effectieve berekenbaarheid' formeel te definiëren omdat de term gebruikt wordt voor de klasse van functies, waarvan de computerexpert denkt dat zij met behulp van een programma kunnen worden uitgerekend. De functie *kwadraat* die als input een geheel getal accepteert, (bijvoorbeeld in decimale notatie) en die als output dat getal met



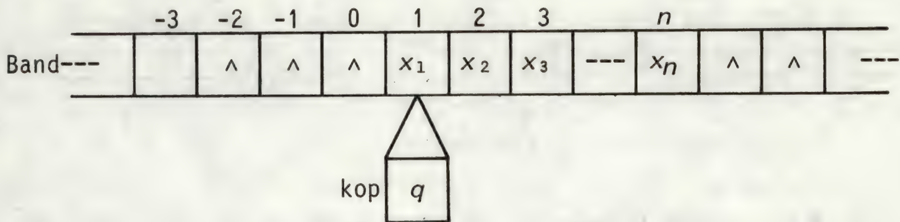
Figuur 0.1 Een naïef computermodel

zichzelf vermenigvuldigd oplevert, is een voorbeeld van een effectief berekenbare functie. Maar wat betekent dat nu in de praktijk? Geen enkele computer kan *kwadraat(x)* berekenen voor alle mogelijke gehele getallen x en wel om de simpele reden dat de opslagcapaciteit van computers eindig is. Dat wil zeggen dat voor voldoende grote waarde van x de waarde *kwadraat(x)* niet in het computergeheugen kan worden opgeslagen. Toch is dit geen echte beperking: als de opslagcapaciteit van onze computer maar groot genoeg is, kunnen we ook van een grote waarde van x het kwadraat berekenen. Bij onze definitie van berekenbaarheid laten we daarom de praktische beperking ten aanzien van opslagcapaciteit buiten beschouwing. We noemen elke functie effectief berekenbaar, die berekenbaar is op een machine met onbegrensde opslagcapaciteit. Als we een model ontwikkelen voor onze 'black box' dan moeten we in dit model dus ook uitgaan van een onbegrensd geheugen. Bij elke eindige berekening wordt weliswaar een eindige hoeveelheid geheugenruimte gebruikt, maar deze geheugenruimte is niet *a priori* begrensd.

Het model dat we zullen gebruiken voor de 'black box' werd ontwikkeld door één van de grondleggers van de berekenbaarheidstheorie Alan Turing (1936). Er bestaan verscheidene andere modellen, zoals Post Machines (Post, 1936) en URIMs (Unlimited Register Ideal Machines, Shepherdson & Sturgis, 1963). De *These van Church*, het eerst geformuleerd in 1936 (Church 1936a) en voortbouwend op het werk van Gödel (1931) en Kleene (1936), kan als volgt onder woorden worden gebracht: elke effectief berekenbare functie is Turing berekenbaar, dat wil zeggen berekenbaar met een Turingmachine. Omdat er geen strenge formele definitie bestaat van het begrip 'effectief berekenbaar' is het onmogelijk deze bewering te bewijzen, maar alles wijst erop dat zij wel degelijk waar is. Ten eerste heeft nog nooit iemand een intuïtief berekenbare functie kunnen bedenken die niet Turing berekenbaar is. Ten tweede berekenen de andere modellen, zoals Post machines en URIMs precies de klasse van Turing berekenbare functies. Tenslotte blijkt elke poging om een schema te ontwikkelen dat alle effectief berekenbare functies voortbrengt, te leiden tot precies de klasse van Turing berekenbare functies (zie hoofdstuk 5). Overal waar men 'Turing berekenbaar' leest kan men dus ook 'effectief berekenbaar' lezen en omgekeerd.

De Turing machine (TM) bezit een onbegrensde opslagcapaciteit in de

vorm van een *band* die is onderverdeeld in een aantal geheugenplaatsen, genummerd met ...-2,-1,0,1,2,..., zoals in Figuur 0.2 is aangegeven. In elke geheugenplaats kan één symbool uit een tevoren gedefinieerd alfabet worden opgeslagen. Aan het begin is de band leeg: elke geheugenplaats bevat het symbool voor 'leeg' dat we met $\wedge$ weergeven. Vervolgens kan een input $x = x_1x_2...x_n$ symbool voor symbool naar de geheugenplaatsen $1,2,...,n$ worden geschreven.



Figuur 0.2 De beginconfiguratie van een Turingmachine met input $x = x_1x_2...x_n$

De TM heeft verder een *leesschrijfkop* die vooruit en achteruit langs de band kan worden bewogen en waarmee symbolen kunnen worden gelezen, geschreven en uitgewist. De leesschrijfkop bevindt zich steeds in één van een eindig aantal toestanden Q. Aan het begin bevindt de kop zich in de *begintoestand* $q_0 \in Q$ en wordt de inhoud van locatie 1 gelezen. De machine stopt alleen als één van de *eindtoestanden* wordt bereikt.

Op ieder moment van de berekening bevindt de kop zich dus in de een of andere toestand q_k en wordt een symbool van de band gelezen (het *huidige tapesymbool*). De volgende stap die door TM wordt uitgevoerd hangt af van de toestand q_k en van het gelezen symbool. Als q_k geen eindtoestand is en de machine dus niet stopt dan kan de kop het huidige tapesymbool overschrijven en vervolgens een locatie naar links of naar rechts bewegen. Tegelijkertijd kan de toestand q_k al dan niet veranderen. Als de machine wel stopt dan worden de symbolen op de locaties $1,2,...,m$ als de output beschouwd. Hierbij is locatie $m+1$ de eerste lege locatie.

Welke actie moet worden uitgevoerd bij elke stap van de berekening wordt bepaald door het *Turingmachine programma*. We zullen, zoals gebruikelijk, het begrip 'Turingmachine' niet alleen gebruiken voor de

machine zelf, maar ook voor het bijbehorende programma. Twee Turingmachines zijn dus verschillend als hun programma's verschillend zijn. Elke stap in een TM programma wordt beschreven als een uit te voeren actie afhankelijk van de toestand van de kop en het gelezen symbool op de tape. De acties worden weergegeven als vijftupels, zoals in Figuur 0.3.



Figuur 0. 3

Het TM programma bestaat uit een lijst van dergelijke vijftupels. Als het goed is staat er voor elke huidige toestand en voor elk mogelijk daarbij horend tapesymbool precies één relevant vijftupel in de lijst, dat de nieuwe toestand van de kop definieert, het symbool dat het huidige tapesymbool moet vervangen aangeeft en de richting bepaalt waarin de kop zich moet bewegen, (L voor links, R voor rechts en 0 voor niet bewegen). Een dergelijk TM programma met unieke instructies heet *deterministisch*.

Ter illustratie construeren wij een heel eenvoudige Turingmachine voor de volgende berekening. We veronderstellen dat de input een string x is, waarin een positief aantal malen het teken 1 voorkomt. De TM moet als output een 1 opleveren als het aantal enen in x even is en anders een 0. De TM berekent dus de functie *even* met

$$even(x) = \begin{cases} 1 & \text{als } x \text{ een even aantal enen bevat} \\ 0 & \text{anders} \end{cases}$$

We formuleren eerst een algoritme in 'gestructureerd Nederlands' en hieruit leiden we vervolgens het TM programma af. Het algoritme berust op het volgende idee: doorloop de inputstring van links naar rechts en houd bij of er een even of oneven aantal enen is geteld door naar de toestand q_0 over te gaan als een even aantal enen is geteld en naar de toestand q_1 als het aantal oneven is. Uiteindelijk komen we bij een lege geheugenplaats op de tape. Als we dan in toestand q_0 zijn is het totaal

aantal enen even en als we in toestand q_1 zijn dan is het oneven. We starten de TM dus met de inputstring op de geheugenplaatsen 1,2,... en in begintoestand q_0, terwijl de leesschrijfkop boven geheugenplaats 1 staat. Het begin van het algoritme kunnen we dan als volgt beschrijven, (de accolades bevatten commentaar):

zolang huidige-tape-symbool = 1 *doe*
  als toestand = q_0 *dan* ga over naar toestand q_1; ga naar rechts
    anders {toestand = q_1} ga over naar toestand q_0; ga naar rechts
  eindeals
eindezolang;
{huidige-tape-symbool = $\wedge$} ga naar links

Vervolgens wissen we alle enen op de tape uit en schrijven we de output op plaats 1. We bewegen daartoe de kop naar links en vervangen elke gelezen 1 door het symbool voor 'leeg' $\wedge$. Dit zetten we voort totdat we een $\wedge$ lezen. Dit betekent dat we op plaats 0 zijn aangeland; vervolgens bewegen we de kop één plaats naar rechts ter voorbereiding van het derde deel van het algoritme: het schrijven van de output. Het tweede deel luidt:

zolang huidige-tape-symbool = 1 *doe*
  vervang symbool door $\wedge$; ga naar links
eindezolang;
{huidige-tape-symbool = $\wedge$ } ga naar rechts

De kop staat nu weer boven plaats 1 en daar schrijven we een 1 als we ons in toestand q_0 bevinden en een 0 als we in toestand q_1 zijn. Tegelijkertijd gaan we over naar de eindtoestand q_2. Het derde deel van het algoritme wordt dan:

{huidige-tape-symbool = $\wedge$}
als toestand = q_0 *dan* ga over naar toestand q_2; vervang symbool door 1
  anders {toestand = q_1} ga over naar toestand q_2; vervang symbool door 0
eindeals

Het eerste deel van het algoritme kan worden vertaald in de volgende rij vijftupels:

$(q_0,1,q_1,1,R)$
$(q_1,1,q_0,1,R)$
$(q_0,\wedge,q_0,\wedge,L)$
$(q_1,\wedge,q_1,\wedge,L)$

Het tweede deel wordt

$(q_0,1,q_0,\wedge,L)$
$(q_1,1,q_1,\wedge,L)$
$(q_0,\wedge,q_0,\wedge,R)$
$(q_1,\wedge,q_1,\wedge,R)$

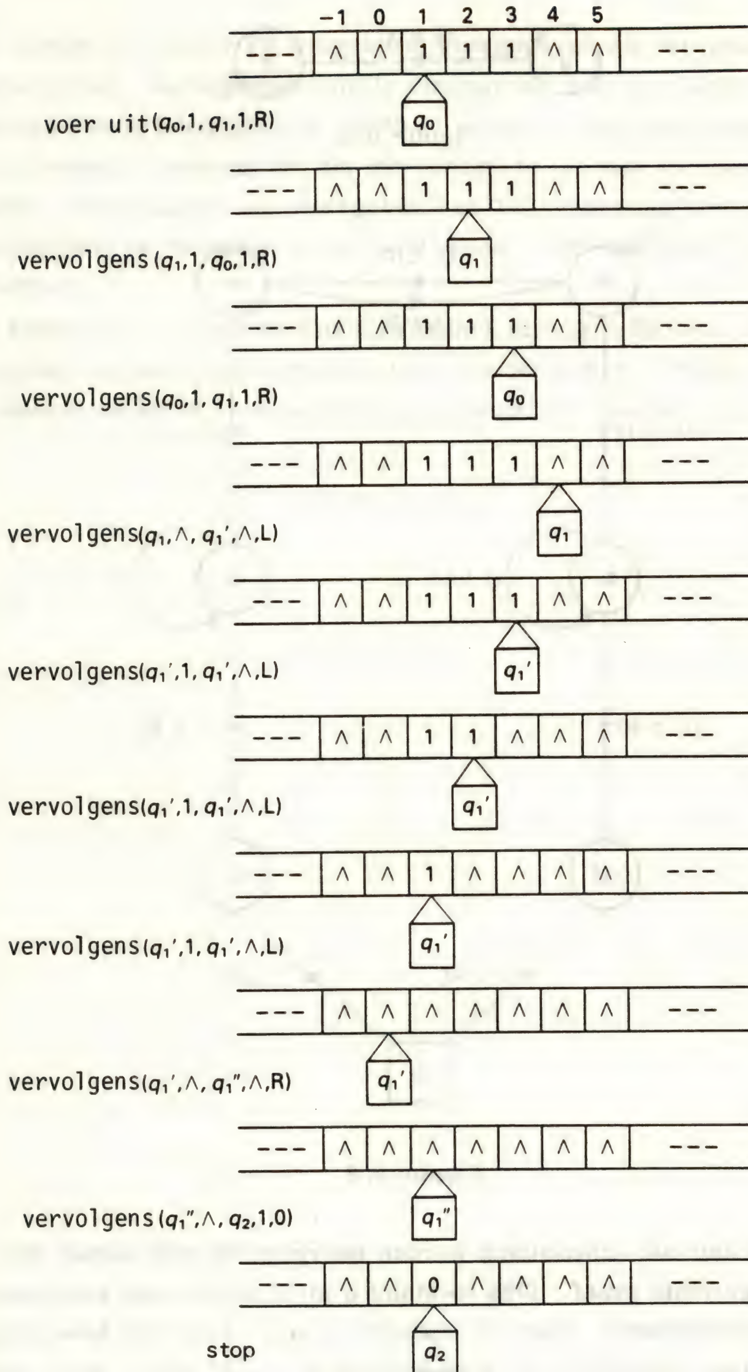
en het derde deel

$(q_0,\wedge,1,0)$
$(q_1,\wedge,0,0)$

Als we het bij deze lijst van vijftupels zouden laten dan zou onze TM niet deterministisch zijn omdat dezelfde toestand met dezelfde input tot verschillende resultaten kan leiden. Dit kan worden opgelost door nog een paar nieuwe toestanden te introduceren. In het eerste deel blijven we q_0 en q_1 gebruiken, maar in het tweede deel gebruiken we q_0' en q_1' en in het derde deel q_0'' en q_1''. Het uiteindelijke programma wordt dan

$(q_0,1,q_1,1,R)$
$(q_1,1,q_0,1,R)$
$(q_0,\wedge,q_0',\wedge,L)$
$(q_1,\wedge,q_1',\wedge,L)$
$(q_0',1,q_0',\wedge,L)$
$(q_1',1,q_1',\wedge,L)$
$(q_0',\wedge,q_0'',\wedge,R)$
$(q_1',\wedge,q_1'',\wedge,R)$
$(q_0'',\wedge,q_2,1,0)$
$(q_1'',\wedge,q_2,0,0)$

hierbij is q_0 de begintoestand van q_2 de enige eindtoestand. Hiermee hebben we laten zien dat *even* een Turing berekenbare functie is. In Figuur 0.4 hebben we alle stappen die de TM uitvoert weergegeven bij input x=111.


-1 0 1 2 3 4 5
voer uit(q0,1,q1,1,R)
vervolgens(q1,1,q0,1,R)
vervolgens(q0,1,q1,1,R)
vervolgens(q1,∧,q1′,∧,L)
vervolgens(q1′,1,q1′,∧,L)
vervolgens(q1′,1,q1′,∧,L)
vervolgens(q1′,1,q1′,∧,L)
vervolgens(q1′,∧,q1″,∧,R)
vervolgens(q1″,∧,q2,1,0)
stop

Figuur 0.4

1

Wiskundige Basisbegrippen

> Maar dit boek kan men niet begrijpen tenzij men heeft geleerd
> de taal te verstaan en de tekens te ontcijferen waarin het is
> geschreven. De taal is de taal der wiskunde ...
> als men die niet kent blijve dit boek gesloten.
>
> Galileo Galilei
> *Il Saggiatore*

In dit hoofdstuk geven we een overzicht van de wiskunde die nodig is om de rest van het boek te kunnen begrijpen. Lezers voor wie dit de eerste kennismaking is met deze stof wordt aangeraden dit hoofdstuk goed te bestuderen, zodat alle begrippen volkomen duidelijk zijn. Als de stof niet nieuw is dan kan het hoofdstuk snel worden doorgelezen zodat de notatie die verder in het boek wordt gebruikt kan worden begrepen.

VERZAMELINGEN

Een *verzameling* bestaat uit een aantal verschillende objecten. Elk object binnen een verzameling wordt een *element* van die verzameling genoemd. Als het aantal elementen in een verzameling niet erg groot is dan kan de verzameling worden beschreven door haar elementen op te sommen. Als bijvoorbeeld D de verzameling van dagen van de week voorstelt dan geldt

$$D = \{\text{Maandag, Dinsdag, Woensdag, Donderdag, Vrijdag, Zaterdag, Zondag}\}$$

De elementen worden opgesomd gescheiden door komma's en het geheel wordt binnen accolades geplaatst. In het algemeen is de volgorde waarin de elementen worden opgesomd niet van belang. We hadden net zo goed kunnen schrijven

D = {Maandag, Woensdag, Vrijdag, Donderdag, Zondag, Dinsdag, Zaterdag}

Als een element x voorkomt in een verzameling A dan schrijven we $x \in A$ (lees: x is element van A) en als x niet in A voorkomt dan schrijven we $x \notin A$. Bijvoorbeeld

Maandag $\in D$

maar

Wasdag $\notin D$

Vaak heeft een verzameling een zeer groot aantal elementen en soms zelfs oneindig veel elementen. Een volledige opsomming is dan wat onhandig en de verzameling kan dan beter worden beschreven door middel van een of andere *definiërende eigenschap* van de elementen. Een element x komt dan in de verzameling voor als x de definiërende eigenschap heeft. Voor de verzameling D is een geschikte definiërende eigenschap

'x is een dag van de week'.

We kunnen dan schrijven

$D = \{x \mid x$ is een dag van de week$\}$

Of om een ander voorbeeld te geven

$P = \{x \mid x$ is een priemgetal$\}$

waarmee een oneindige verzameling van gehele getallen wordt gedefinieerd.

Als we verzamelingen op deze manier definiëren dan moeten we er wel aan denken ook het *universum* waaruit de elementen stammen te beschrijven. Als bijvoorbeeld

$X = \{x \mid x > 2\}$

dan kan de precieze aard van X alleen worden vastgesteld als we ook weten welke waarden x kan aannemen. De verzameling waarbij x alleen gehele waarden kan aannemen is bijvoorbeeld duidelijk verschillend van die waarbij x alle mogelijke reële waarden mag aannemen. In het eerste geval

$2 \circ 1 \notin X$ en in het tweede geval $2 \cdot 1 \in X$. Bij alle verzamelingen die we bekijken komen de elementen voort uit een of ander specifiek universum $\mathcal{U}$. Bijvoorbeeld de gehele getallen, de positieve reële getallen, enzovoorts.

Een heel belangrijke verzameling is de *lege verzameling*. Deze verzameling bevat geen elementen en wordt weergegeven door $\emptyset$ of ook wel door $\{\}$.

A is een deelverzameling van B, notatie $A \subseteq B$, als ieder element van A ook element van B is. Als A geen deelverzameling van B is kunnen we dat aangeven met $A \nsubseteq B$. Dus

$$\{1,2,4\} \subseteq \{1,2,3,4,5\}$$

maar

$$\{2,4,6\} \nsubseteq \{1,2,3,4,5\}$$

Altijd geldt

$$A \subseteq \mathcal{U}$$

en

$$\emptyset \subseteq A \text{ voor alle verzamelingen } A.$$

Twee verzamelingen A en B zijn *gelijk*, notatie $A = B$, als $A \subseteq B$ en $B \subseteq A$. Dus

$$\{1,2,3,4\} = \{2,1,4,3\}$$

maar

$$\{1,2,3,4\} \neq \{2,1,3,5\}$$

Duidelijk is dat $A = B$ desd (dan en slechts dan) geldt als A en B precies dezelfde elementen bevatten. Als $A \subseteq B$; maar $A \neq B$ dan schrijven we $A \subset B$ en noemen we A een *echte deelverzameling* van B. De verzameling van alle deelverzamelingen van een verzameling A heet de *machtverzameling* van A en wordt weergegeven door 2^A. Als bijvoorbeeld $A = \{1,2,3\}$ dan is $2^A = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$.

Operaties op verzamelingen zijn de unaire operatie complement ($'$) en de binaire operaties vereniging ($\cup$); doorsnede ($\cap$) en verschil ($\setminus$). (Een operatie heet unair als hij op één operand wordt toegepast en binair als er twee operanden voor nodig zijn.) De operaties worden als volgt gedefinieerd:

Als A en B verzamelingen zijn dan

$A' = \{x \mid x \notin A\}$

deze verzameling bestaat uit alle elementen van het universum die niet in A voorkomen;

$A \cup B = \{x \mid x \in A \text{ of } x \in B\}$

alle elementen die in A of in B voorkomen;

$A \cap B = \{x \mid x \in A \text{ en } x \in B\}$

alle elementen die zowel in A als in B voorkomen;

$A \setminus B = \{x \mid x \in A \text{ maar } x \notin B\}$

alle elementen uit A die niet in B voorkomen.

Als bijvoorbeeld $\mathcal{U} = \{0,1,2,3,4,5,6,7,8,9\}$, $A = \{0,1,3,5\}$ en $B = \{2,3,5\}$ dan

$A' = \{2,4,6,7,8,9\}$ $\quad A \cap B = \{3,5\}$

$A \cup B = \{0,1,2,3,5\}$ $\quad A \setminus B = \{0,1\}$ $\quad B \setminus A = \{2\}$.

$\cup$ en $\cap$ zijn beide *associatieve* operaties, dat wil zeggen

$(A \cup B) \cup C = A \cup (B \cup C)$

en $(A \cap B) \cup C = A \cap (B \cap C)$ voor alle verzamelingen A, B en C.

De verschiloperatie is niet associatief. $\cup$ en $\cap$ zijn ook *commutatief*, dat wil zeggen

$A \cup B = B \cup A$

en $A \cap B = B \cap A$ voor alle verzamelingen A en B. $\setminus$ is niet commutatief. In stelling 1.1 zijn deze en enkele andere eigenschappen samengevat.

Stelling 1.1 (Eigenschappen van verzamelingen)

Voor alle verzamelingen A, B en C in het universum $\mathcal{U}$ geldt:

(1) Associativiteit $(A \cup B) \cup C = A \cup (B \cup C)$ en
$(A \cap B) \cap C = A \cap (B \cap C)$

(2) Commutativiteit $A \cup B = B \cup A$ en $A \cap B = B \cap A$

(3) Complementariteit $A \cup A' = \mathcal{U}$ en $A \cap A' = \emptyset$

(4) Gelijkmachtigheid $A \cup A = A$ en $A \cap A = A$

(5) Identiteit $A \cup \emptyset = A$ en $A \cap \mathcal{U} = A$

(6) Nul-eigenschap $A \cup \mathcal{U} = \mathcal{U}$ en $A \cap \emptyset = \emptyset$

(7) Involutie-eigenschap $(A')' = A$

(8) Wetten van de Morgan $(A \cup B)' = A' \cap B'$ en $(A \cap B)' = A' \cup B'$

(9) Distributiviteit $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ en
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Omdat vereniging associatief is kunnen we $A \cup B \cup C$ schrijven. Het resultaat is onafhankelijk van de volgorde van het uitvoeren van de operaties altijd hetzelfde Deze zelfde redenering geldt voor $A_1 \cup A_2 \cup \ldots \cup A_n$ als $A_1, A_2, \ldots, A_n$ verzamelingen zijn. Deze laatste vereniging zullen we schrijven als

$$\bigcup_{i=1}^{n} A_i$$

en we schrijven

$$\bigcap_{i=1}^{n} A_i \quad \text{voor} \quad A_1 \cap A_2 \cap \ldots \cap A_n$$

We noemen twee verzamelingen A en B *disjunct* als zij geen elementen gemeenschappelijk hebben. Dat kunnen we ook aangeven door $A \cap B = \emptyset$.

Als A_1 en A_2 verzamelingen zijn dan is hun *produkt* $A_1 \times A_2$ gedefinieerd als de verzameling van alle paren (a_1, a_2) met a_1 uit A_1 en a_2 uit A_2. Dat wil zeggen

$$A_1 \times A_2 = \{(a_1, a_2) | a_1 \in A_1, a_2 \in A_2\}$$

Als bijvoorbeeld $A_1 = \{0,1\}$ en $A_2 = \{x,y,z\}$ dan is $A_1 \times A_2 = \{(0,x),(0,y),(0,z),(1,x),(1,y),(1,z)\}$.

Evenzo bevat $A_1 \times A_2 \times A_3$ alle tripels (a_1, a_2, a_3) met a_1 uit A_1, a_2 uit A_2 en a_3 uit A_3. De definitie kan worden uitgebreid naar alle n-tupels $(a_1, a_2, \ldots, a_n)$ met $a_i \in A_i$ voor $i = 1, 2, \ldots, n$.

Merk op dat $A \times B \neq B \times A$, tenzij $A = B$. De volgende distributieve wetten kunnen gemakkelijk worden aangetoond.

Stelling 1.2

(1) $A \times (B_1 \cup B_2) = (A \times B_2) \cup (A \times B_2)$
en $(A_1 \cup A_2) \times B = (A_1 \times B) \cup (A_2 \times B)$;

(2) $A \times (B_1 \cap B_2) = (A \times B_1) \cap (A \times B_2)$
en $(A_1 \cap A_2) \times B = (A_1 \times B) \cap (A_2 \times B)$;

(3) $A \times (B_1 \setminus B_2) = (A \times B_1) \setminus (A \times B_2)$
en $(A_1 \setminus A_2) \times B = (A_1 \times B) \setminus (A_2 \times B)$.

CARDINALITEIT EN AFTELBAARHEID

A heet een *eindige verzameling* als A een eindig aantal elementen bezit en A heet een *oneindige verzameling* als het aantal elementen in A oneindig is.

De *cardinaliteit* van een eindige verzameling A noteren we als $\#(A)$ en is gedefinieerd als het aantal elementen in A. Als D bijvoorbeeld de verzameling van dagen van de week is dan is $\#(D) = 7$. Een verzameling met cardinaliteit één wordt wel een *singleton* (spreek uit singelton) genoemd.

Bij oneindige verzamelingen zou het handig zijn als we een systematische methode hadden voor het opsommen van de elementen, zodat we over het 'eerste element', het 'tweede element' enzovoort zouden kunnen spreken. Als N bijvoorbeels de verzameling van de natuurlijke getallen is dan begint die opsomming met 1,2,3,4,... en kunnen we het hebben over 'het i-de natuurlijke getal'. Het ligt niet direct voor de hand (sterker nog het is niet waar) dat elke oneindige verzameling zo opgesomd kan worden.

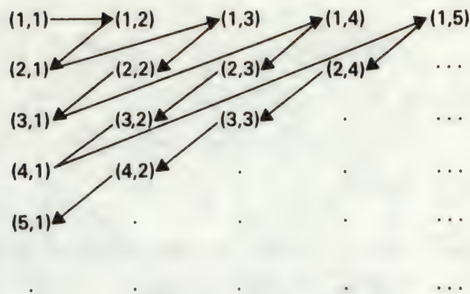
Kunnen we bijvoorbeeld ook Z de verzameling van alle gehele getallen systematisch opsommen? Dat kan inderdaad: we beginnen met 0 en vermelden vervolgens de natuurlijke getallen in opklimmende grootte twee keer met verschillend teken. Dus

$$0, +1, -1, +2, -2, +3, -3, \ldots$$

Uiteindelijk bereiken we zo elke $z \in Z$.

In Figuur 1.1 hebben we een manier weergegeven om alle elementen uit $N \times N$ systematisch op te sommen. Hierbij wordt (m,n) genoemd voor (m',n') als ofwel $m+n < m'+n'$, ofwel als $m+n = m'+n'$ en $m < m'$.

Steeds als we een systematische manier vinden om alle elementen van



Figuur 1.1

een verzameling op te sommen, kunnen we spreken van het 'i-de element van de verzameling' en we noemen de verzameling dan *aftelbaar*. Elke eindige verzameling is aftelbaar maar dit geldt niet voor elke oneindige verzameling. Een voorbeeld van een niet aftelbare oneindige verzameling is $\mathbb{R}$ - de verzameling van de reële getallen. Sterker nog: we zullen aantonen dat de reële getallen $\geqslant 0$ en < 1 niet aftelbaar zijn. We geven dit interval aan met $[0,1)$. Het bewijs maakt gebruik van de *diagonalisatiemethode van Cantor*. We veronderstellen dat we beschikken over een systematische opsomming van de elementen en we zullen vervolgens aantonen dat deze veronderstelling tot een tegenspraak leidt en dus niet juist kan zijn. Als we de getallen in decimale notatie weergeven dan ziet onze opsomming er zo uit

1-ste getal $0 \cdot a_{11} a_{12} a_{13} \ldots$

2-de getal $0 \cdot a_{21} a_{22} a_{23} \ldots$

3-de getal $0 \cdot a_{31} a_{32} a_{33} \ldots$

hierbij stelt elke a_{ij} dus één der cijfers 0,1,2,3,4,5,6,7,8 en 9 voor. In deze opsomming zal dus uiteindelijk elke reële waarde uit het interval $[0,1)$ moeten voorkomen. We gaan nu uit van het getal

$$0 \cdot a_{11} a_{22} a_{33} \ldots$$

samengesteld uit de cijfers op de diagonaal van onze lijst en we definiëren een nieuw getal

$$0 \cdot b_{11} b_{22} b_{33} \ldots$$

met

$$b_{ii} = \begin{cases} a_{ii} + 1 & \text{als } a_{ii} < 9 \\ 0 & \text{als } a_{ii} = 9 \end{cases}$$

Dit laatste getal nu kan nooit in onze lijst voorkomen omdat het immers van het i-de getal in de lijst verschilt in minstens het i-de cijfer, ($i = 1,2,3,\ldots$). Onze lijst is dus toch niet volledig en dit is in tegenspraak met onze veronderstelling dat dit wel het geval was. Hiermee is aangetoond dat $[0,1)$ niet aftelbaar is en de verzameling $\mathbb{R}$ van de reële getallen is dat dus zeker niet.

Voor eindige verzamelingen geldt dat $\#(A_1 \times A_2) = \#(A_1) \times \#(A_2)$ en met behulp van Figuur 1.1 kan worden aangetoond dat $A_1 \times A_2$ aftelbaar is als zowel A_1 als A_2 aftelbaar is. Dit resultaat kan worden uitgebreid tot: voor elke $n > 1$ geldt dat $\#(A_1 \times A_2 \times \ldots \times A_n) = \#(A_1) \times \#(A_2) \times \ldots$ $\ldots \times \#(A_n)$ als de verzamelingen eindig zijn en dat $A_1 \times A_2 \times \ldots \times A_n$ aftelbaar is als $A_1, A_2, \ldots, A_n$ aftelbaar zijn.

RELATIES

Een *relatie* R is elke deelverzameling van $A_1 \times A_2$, waarbij A_1 het *domein* of definitiegebied en A_2 het *bereik* (Engels: range) of beeld van R. Bij veel relaties zijn domein en beeld dezelfde verzameling, bijvoorbeeld A. In dit geval noemen we $R \subseteq A \times A$ een relatie *op* A. Als A bijvoorbeeld de verzameling $\{0,1,2,3\}$ is dan is de verzameling van geordende paren

$$L = \{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$$

de relatie op A die overeenkomt met 'strikt kleiner dan' en

$$E = \{(0,0),(1,1),(2,2),(3,3)\}$$

is de relatie op A overeenkomend met 'gelijk aan'. We zullen in het vervolg aRb gebruiken voor $(a,b) \in R$ en $a \not R b$ voor $(a,b) \notin R$.

Een relatie R op A heet *reflexief* als

aRa voor alle $a \in A$.

De relatie L is dus geen reflexieve relatie op A omdat $(0,0) \notin L$, maar de relatie E is wel reflexief.

Een relatie is *symmetrisch*

als uit aRb volgt dat bRa.

L is niet symmetrisch omdat $(0,1) \in L$, maar $(1,0) \notin L$. E is wel symmetrisch.

Een relatie R op A is *transitief*

als uit aRb en bRc volgt dat aRc.

L en E zijn beide transitieve relaties.

Een relatie is *antisymmetrisch*

als uit aRb en bRa volgt dat $a = b$.

E en L zijn antisymmetrisch en dit geldt ook voor $E \cup L$. De relatie $\{(0,1),(1,0)\}$ is niet antisymmetrisch.

Als een relatie R op A reflexief, symmetrisch en transitief is dan heet de relatie een *equivalentierelatie*. Uit het voorgaande blijkt dat E een equivalentierelatie is. Een ander voorbeeld is

$$V = \{(0,0),(0,2),(1,1),(1,3),(2,2),(2,0),(3,1),(3,3)\}$$

Als R een equivalentierelatie is op A en $a \in A$ dan kunnen we de verzameling van alle elementen $\bar{a}$ beschouwen die via R aan a gerelateerd zijn. Dat wil zeggen

$$\bar{a} = \{b \in A \mid aRb\}$$

Een dergelijke verzameling wordt een *equivalentieklasse* genoemd.

Bij E bestaan vier verschillende equivalentieklassen $\bar{0} = \{0\}$, $\bar{1} = \{1\}$, $\bar{2} = \{2\}$, $\bar{3} = \{3\}$. Bij V bestaan maar twee verschillende equivalentieklassen $\bar{0} = \bar{2} = \{0,2\}$ en $\bar{1} = \bar{3} = \{1,3\}$. Hieruit blijkt dat de equivalentieklassen die worden gedefinieerd door een equivalentierelatie R op een verzameling A deze verzameling altijd verdelen in een aantal disjuncte niet-lege verzamelingen, (zie oefening 1.6).

Als R een willekeurige relatie op A is dan is de *reflexieve* (*respectievelijk symmetrische* of *transitieve*) *insluiting* van R de kleinste reflexieve

(symmetrische of transitieve) relatie op A waar R een deelverzameling van is. Als bijvoorbeeld

$$R = \{(0,1),(1,1)(1,2)\}$$

een relatie op $A = \{0,1,2\}$ is dan is de bijbehorende reflexieve insluiting

$$\{(0,0),(0,1),(1,1),(1,2),(2,2)\}$$

de symmetrische insluiting is

$$\{(0,1),(1,0),(1,1),(1,2),(2,1)\}$$

en de transitieve insluiting is

$$\{(0,1),(0,2)(1,1),(1,2)\}$$

Zo heeft de relatie $<$ gedefinieerd op alle gehele getallen als reflexieve insluiting de relatie $\leqslant$, als symmetrische insluiting de relatie $\neq$ en omdat de relatie $<$ zelf transitief is, is de transitieve insluiting de relatie zelf.

Een relatie op A is een *ordeningsrelatie* op A als de relatie reflexief, antisymmetrisch en transitief is. Als een dergelijke relatie bestaat voor een niet-lege verzameling A dan heet A *partieel geordend* ten opzichte van die relatie. Als een verzameling A partieel geordend is ten opzichte van een relatie $\subseteq$ dan kunnen er in A de elementen a en b voorkomen met $a \nsubseteq b$ en $b \nsubseteq a$. Dergelijke elementen worden niet-vergelijkbaar genoemd. Een partieel geordende verzameling heet *totaal geordend* als de verzameling geen niet-vergelijkbare elementen bevat. De verzameling der gehele getallen Z is een voorbeeld van een totaal geordende verzameling ten opzichte van de relatie $\leqslant$. De machtverzameling van een verzameling is een voorbeeld van een partieel, maar niet totaal geordende verzameling ten opzichte van de relatie $\subseteq$.

Op elke aftelbare verzameling A kunnen we een totale ordening aanbrengen door de elementen $a_1, a_2, \ldots$ in een bepaalde volgorde op te sommen en te stellen dat $a_i \subset a_j$ desd $i \leqslant j$.

Als $\subseteq$ een ordeningsrelatie op A aangeeft dan zullen we $\subset$ gebruiken voor $a \subseteq b$ en $a \neq b$. Dit komt overeen met de bekende ordeningsrelaties $<$ en $\leqslant$ op verzamelingen van getallen en met $\subset$ en $\subseteq$ bij verzamelingen.

FUNCTIES

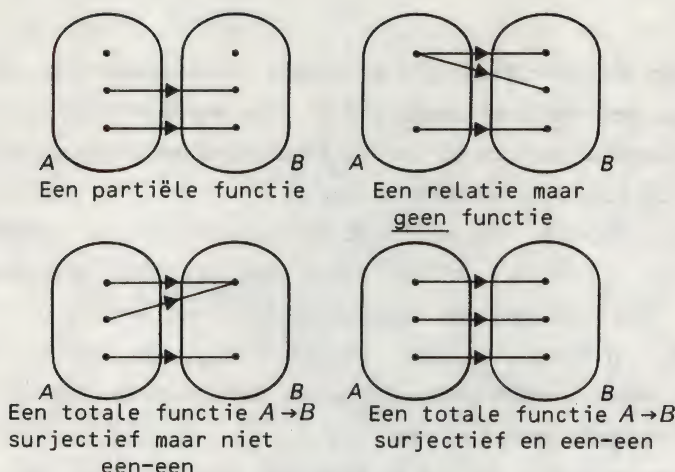
Een functie f van een verzameling A naar een verzameling B verbindt met elke $a \in A$ een uniek element $b \in B$. Dit wordt weergegeven als $b = f(a)$. Vaak bestaat er een of andere formule of rekenkundige uitdrukking voor de functie f in termen van a.

We kunnen een functie $f{:}A \to B$ ook beschouwen als de verzameling van alle paren (a,b) met $a \in A$ en $b \in B$ en $b = f(a)$. Een functie $f{:}A \to B$ wordt dus soms ook wel gedefinieerd als een relatie in $A \times B$ met de eigenschap dat als $(a,b) \in f$ en $(a,c) \in f$ dat dan noodzakelijkerwijs $b = c$. Deze laatste restrictie is de vertaling van de eis dat $f(a)$ een unieke waarde moet hebben.

Als f een functie van A naar B is dan schrijven we, zoals we hierboven al zagen, $f{:}A \to B$ en we noemen A het *domein* van f en B het *codomein* van f. De verzameling $\{f(a) | a \in A\}$ heet het *bereik* van f en dit is een deelverzameling van het codomein van f. De functie *dubbel*: $N \to N$, gedefinieerd als $dubbel(n) = 2n$ voor alle $n \in N$ heeft bijvoorbeeld als bereik de verzameling van de even getallen.

Meestal is een functie $f{:}A \to B$ gedefinieerd voor alle $a \in A$. Deze functies worden wel *totale functies* genoemd. Soms zullen we een functie $f{:}A \to B$ gebruiken waarbij $f(a)$ niet voor alle $a \in A$ is gedefinieerd. Zo'n functie heet een *partiële functie*. De functie *reciproke*: $\mathbb{R} \to \mathbb{R}$ gedefinieerd door $reciproke(x) = 1/x$ bijvoorbeeld is partieel omdat deze niet is gedefinieerd voor $x = 0$. Bij elke partiële functie kan een totale functie worden geconstrueerd door het domein geschikt te kiezen. De functie *reciproke* is bijvoorbeeld wel een totale functie op $\mathbb{R} \setminus \{0\} \to \mathbb{R}$. De lezer kan er in het vervolg van uitgaan dat een functie totaal is tenzij het tegendeel expliciet wordt vermeld.

Een totale functie $f{:}A \to B$ is *surjectief* als er voor elke $b \in B$ een $a \in A$ bestaat met $f(a) = b$. De functie is *injectief* of *een-op-een* als bij elk element van A een uniek element van B hoort, dat wil zeggen dat $f(a_1) = f(a_2)$ desd als $a_1 = a_2$. In Figuur 1.2 hebben we een aantal gevallen weergegeven door de elementen $(a,b) \in f$ aan te geven met een pijl van $a \in A$ naar $b \in B$. Als een totale functie f zowel surjectief als injectief is dan noemen we f een *bijectie*. Met behulp van dit begrip kunnen we nu *aftelbaarheid* formeel definiëren als: een verzameling A is aftelbaar als deze eindig is of als er een bijectie $f{:}A \to N$



Figuur 1.2

bestaat. Deze bijectie komt neer op de afbeelding van het i-de element van A op het positieve gehele getal i.

Als x een reëel getal is dan is $\lceil x \rceil$ het kleinste gehele getal groter dan x en is $\lfloor x \rfloor$ het grootste gehele getal kleiner dan x. De functies $rondop(x) = \lceil x \rceil$ en $rondaf(x) = \lfloor x \rfloor$ van $\mathbb{R}$ naar Z zijn beiden surjectief, maar niet injectief. De functie $kwadraat(n) = n^2$ van N op N is een voorbeeld van een injectieve functie die niet surjectief is. Zouden we het domein van de functie uitbreiden tot Z dan was de functie ook niet meer injectief.

De eenheidsfunctie $identiek(a) = a$ is een bijectie $A \to A$ voor elke verzameling A. Interessanter is de bijectie gedefinieerd door $wissel: N \to N$ met

$$wissel(n) = \begin{cases} n-1 & \text{als } n \text{ even is} \\ n+1 & \text{als } n \text{ oneven is.} \end{cases}$$

Als het codomein van een functie wordt gevormd door de verzameling van logische of Boole'se waarden {T,F} (T staat voor True of waar en F voor False of onwaar), dan wordt de functie een *predicaat* genoemd. De functie $even: N \to \{T, F\}$ gedefinieerd door

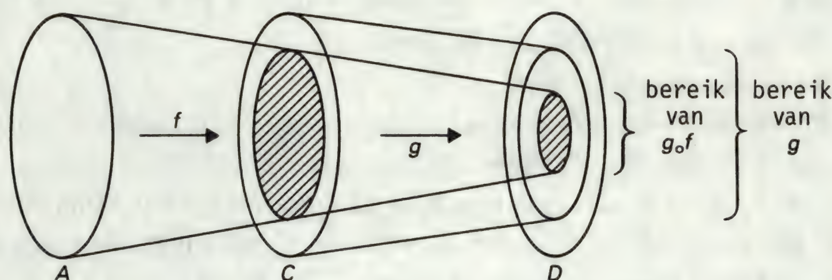
$$even(n) = \begin{cases} T & \text{als } n \text{ even is} \\ F & \text{als } n \text{ oneven is} \end{cases}$$

is een voorbeeld van een predicaat op de natuurlijke getallen.

Tot nu toe bekeken we alleen functies van één variabele of unaire functies. Functies van twee variabelen, de eerste uit een verzameling A_1 en de tweede uit een verzameling A_2, worden *binaire* functies genoemd. Algemeen heet een functie $f:A_1 \times A_2 \times \ldots \times A_n \rightarrow B$ een n-aire functie. Het aantal argumenten of variabelen van de functie wordt de *ariteit* van de functie genoemd.

Bijvoorbeeld $som:\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ is een binaire functie met als definitie $som(x,y) = x+y$. *Tripelsom* is een 3-aire functie $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ met als definitie $tripelsom(x,y,z) = x+y+z$.

Stel $f:A \rightarrow B$ en $g:C \rightarrow D$ zijn twee functies. Als het bereik van f een deelverzameling is van het domein van g dan kunnen we spreken van de samengestelde functie $g_o f$, waarbij $g\ f:A \rightarrow D$ is gedefinieerd als $g_o f(a) = g(f(a))$ voor alle $a \in A$. Het bereik van de functie $g_o f$ is een deelverzameling van het bereik van g, (zie Figuur 1.3).



Figuur 1.3

Als bijvoorbeeld $succ:N \rightarrow N$ is gedefinieerd als $succ(n) = n+1$ dan is $succ_o kwadraat:N \rightarrow N$ gedefinieerd door $succ_o kwadraat(n) = succ(kwadraat(n)) = succ(n^2) = n^2+1$, maar $kwadraat_o succ(n) = (n+1)^2$. Hiermee is duidelijk dat het samenstellen van functies in het algemeen geen commutatieve operatie is. Wel is er altijd sprake van associativiteit.

Als $f:A \rightarrow B$ en $f^{-1}:B \rightarrow A$ zodanige functies zijn dat $f^{-1}{}_o f:A \rightarrow A$ en $f_o f^{-1}:B \rightarrow B$ beiden de identiteitsfunctie zijn dan wordt f^{-1} de *inverse* functie van f genoemd. De inverse van een functie $f:A \rightarrow B$ is alleen dan een totale functie $B \rightarrow A$ als f een bijectie is, (zie Oefening 1.12). De inverse van de identiteitsfunctie *identiteit*$:A \rightarrow A$ is de functie *identiteit* zelf. Dit is echter niet de enige functie die zijn eigen inverse is; ook de eerder gedefinieerde functie *wissel*(n) is zo'n functie. We geven nog enkele voorbeelden van functies die inversen hebben:

somk:R→R($k \neq 0$) gedefinieerd door *somk*(*k*) $= x + k$ heeft als inverse *subk*:R→R met *subk*(*x*) $= x - k$. De functie *multk*:R→R ($k \neq 0$, $k \neq 1$) gedefinieerd door *multk*(*x*) $= x * k$ heeft als inverse *divk*:R→R met *divk*(*x*) $= x/k$.

Als f en g beiden (mogelijk partiële) functies zijn $A \to B$ dan zeggen we f is gelijk aan g of $f = g$ voor alle $x \in A$ als ofwel $f(x)$ en $g(x)$ beiden niet gedefinieerd zijn, ofwel $f(x) = g(x)$. Dus bijvoorbeeld: $som2 = som1 \circ som1$ en $som3 = sub1 \circ som4 = som1^{-1} \circ som4$ enzovoort.

INDUCTIE EN RECURSIE

Inductie is een krachtige techniek die kan worden toegepast telkens als men een aftelbare rij stellingen $\{T_n | n \in N\}$ wil bewijzen. We willen bijvoorbeeld bewijzen dat $8^n - 3^n$ deelbaar is door 5 voor $n = 1,2,3,\ldots$.

Dit bewijs gaat als volgt in zijn werk:

(1) We bewijzen dat T_1 waar is

(2) We bewijzen dat voor alle $k \in N$ geldt dat als T_n waar is voor alle $n < k$ dat dan T_k waar is.

Als nu T_1 waar is dan volgt wegens (2) dat T_2 waar is. Opnieuw wegens (2) volgt nu dat T_3 waar is, enzovoort. We zullen aantonen dat inductie inderdaad een juiste bewijstechniek is. Veronderstel dat $\{T_n | n \in N\}$ een aftelbare rij stellingen is met de volgende eigenschappen: (a) T_1 is waar, (b) voor alle $k \in N$ geldt dat als T_n waar is voor alle $n < k$ dan is T_k waar; dan volgt hieruit dat T_n waar is voor alle $n \in N$. Immers als dit niet het geval was dan bestond er een $j > 0$ met T_j onwaar, maar T_n is waar voor alle $n < j$. Nu geldt $j \neq 1$ wegens (a), maar nu leidt de veronderstelling tot een tegenspraak met (b) in het geval $k = j$.

Bij een bewijs met inductie moet men ervoor zorgen dat dit correct wordt opgebouwd. Het schema op bladzijde 15 kan daartoe worden gebruikt. Als voorbeeld zullen we bewijzen dat $8^n - 3^n$ deelbaar is door 5 voor alle $n \in N$. Het bewijs wordt gegeven met behulp van inductie. De bewering is waar voor $n = 1$ omdat $8^1 - 3^1 = 5$ deelbaar is door 5. Veronderstel dat de bewering geldt voor alle $n < k$ dat wil zeggen dat $8^n - 3^n$ deelbaar is door 5 als $n < k$.

Het bewijs wordt gegeven met behulp van inductie.	de eerste regel
De bewering is waar voor $n = 1$ omdat ...	bewijs dat T_1 waar is
Veronderstel dat de bewering waar is voor alle $n < k$, dat wil zeggen dat ...	de *inductiehypothese*
Dan ...	bewijs dat T_k waar is volgt uit inductie-hypothese
Dus geldt de bewering voor $n = k$.	
Op grond van inductie geldt de bewering dus voor alle $n \in N$.	de laatste regel

Nu is $8^k - 3^k = 8 \times 8^{k-1} - 3 \times 3^{k-1} = 5 \times 8^{k-1} + 3 \times (8^{k-1} - 3^{k-1})$

en wegens de inductiehypothese is $8^{k-1} - 3^{k-1}$ deelbaar door 5. Omdat ook $5 \times 8^{k-1}$ deelbaar is door 5 volgt dat $8^k - 3^k$ deelbaar is door 5. De bewering is dus waar voor $n = k$.
Op grond van inductie geldt de bewering dus voor alle $n \in N$.

Tussen inductie en recursie bestaat een nauw verband. Recursie is een methode van definiëren van een functie f met behulp van een expressie die zelf f weer bevat. Een veel gebruikt voorbeeld is de functie $fac: N \rightarrow N$ met

$$fac(n) = \textit{als } n = 1 \textit{ dan } 1$$
$$\textit{anders } n \times fac(n-1)$$

Als we bijvoorbeeld $fac(3)$ met behulp van deze definitie uitrekenen dan krijgen we

$$\begin{aligned} fac(3) &= 3 \times fac(2) \\ &= 3 \times 2 \times fac(1) \\ &= 3 \times 2 \times 1. \end{aligned}$$

Als we $n!$ definiëren als $n \times (n-1) \times (n-2) \times \ldots \times 1$ dan kunnen we bewijzen dat $fac(n) = n!$ voor alle $n \in N$. Waarschijnlijk is het geen verrassing meer dat dit bewijs met inductie gaat. Zeker is $fac(1) = 1 = 1!$,

dus de bewering geldt voor $n = 1$. Als we veronderstellen dat $fac(n) = n!$ voor alle $n < k$ dan is $fac(k) = k \times fac(k-1) = k \times (k-1)! = k!$ dus geldt de bewering voor $n = k$. Op grond van inductie geldt de bewering dus voor alle $n \in N$.

We geven nog een voorbeeld van een recursieve functie. De functie $som: N \times N \to N$ wordt gedefinieerd door

$$
\begin{aligned}
som(m,n) = \ & als\ m=1\ dan \\
& \qquad als\ n=1\ dan\ 2 \\
& \qquad anders\ succ(som(m,n-1)) \\
& anders\ succ(som(m-1,n))
\end{aligned}
$$

Na enig proberen blijkt dat $som(m,n) = m+n$ voor alle $m,n \in N$, maar een formeel bewijs met behulp van inductie vergt enige nauwgezetheid. We hebben hier te maken met twee argumenten m en n en de aftelling van de stellingen moet dus gebeuren op $N \times N$. Op $N \times N$ is een ordening gedefinieerd door

$(m,n) \subset (m',n')$ desd
$m+n < m' + n'$ of als $m+n = m' + n'$ dan $m < m'$.

Met behulp van deze ordening kunnen we een bewijs met inductie uitvoeren. De bewering geldt voor het eerste element in de ordening $(1,1)$ omdat $som(1,1) = 2 = 1+1$. Veronderstel dat $(j,k) \neq (1,1)$ en de bewering geldt voor alle $(m,n) \subset (j,k)$ dan

$$
\begin{aligned}
som(j,k) = \ & als\ j+1\ dan\ succ(som(j,k-1)) \\
& anders\ succ(som(j-1,k))
\end{aligned}
$$

Nu geldt $(j,k-1) \subset (j,k)$ en $(j-1,k) \subset (j,k)$ en dus is op grond van de inductiehypothese $som(j,k-1) = som(j-1,k) = j+k-1$. Dus is $som(j,k) = som(j+k-1) = j+k$ voor $j=1$ en voor $j>1$. De bewering geldt dus voor (j,k) en op grond van inductie dus voor alle $m,n \in N$.

STRINGS

Met een *string* bedoelen we een eindige rij symbolen $a_1 a_2 \ldots a_n$, waarbij elke a_i een keuze is uit een of ander eindig alfabet Σ. In een string

mag hetzelfde symbool meer dan één keer voorkomen. Een voorbeeld van een string uit het alfabet $\Sigma = \{0,1\}$ is 001110. Het aantal symbolen in een string noemen we de *lengte* van die string. De string 001110 heeft lengte 6. De *lege string* geven we aan met een ε. Deze string bevat geen symbolen en heeft dus lengte 0. We zullen de lengte van een string x aangeven met $|x|$. Dus $|001110| = 6$ en $|\varepsilon| = 0$.

De wiskunde die op strings van toepassing is, is heel eenvoudig. Eerst moeten we een *alfabet* specificeren. Dit is een niet-lege eindige verzameling van symbolen die in onze strings kunnen verschijnen. Als het lege symbool of de spatie in de strings moet kunnen voorkomen, dan moet dit symbool ook in ons alfabet voorkomen. Omdat 'niets' zo lastig weer te geven is gebruiken we voor het lege symbool het teken $\wedge$. We schrijven dus $00\wedge11\wedge01$ in plaats van 00 11 01. Merk op dat het lege symbool zelf een string vormt met lengte 1 - het moet dus niet verward worden met de lege string.

Als we een alfabet Σ hebben gedefinieerd dan kunnen we vervolgens de verzameling van alle mogelijke eindige strings over het alfabet Σ aangeven door Σ^*. Σ^* heeft altijd een aftelbaar oneindig aantal elementen (zie Oefening 1.4). Als bijvoorbeeld $\Sigma = \{0,1\}$ dan is een mogelijke ordening van Σ^*:

$$\varepsilon,0,1,00,01,10,11,000,001,010,...$$

Deze ordening van de elementen van $\{0,1\}^*$ wordt de *lexicografische ordening* genoemd. We kunnen de gebruikelijke notatie uit de verzamelingenleer gebruiken $x \in \Sigma^*$ om aan te geven dat x een element van Σ^* is en $x \notin \Sigma^*$ als dat niet het geval is. Uit de definitie volgt dat in ieder geval $\varepsilon \in \Sigma^*$ voor alle verzamelingen Σ.

Als $x \in \Sigma^*$ een string met lengte m is dan kunnen we die weergeven met $x = a_1a_2...a_m$, waarbij $a_i \in \Sigma$, $1 \leqslant i \leqslant m$. Als $x \in \Sigma^*$ een string ter lengte m is en $y \in \Sigma^*$ een string ter lengte n is dan wordt de *concatenatie* van x en y weergegeven door xy. Deze is gedefinieerd als een string met lengte $m+n$, waarbij de eerste m symbolen overeenkomen met de string x en de volgende n met y. Als dus $x = a_1a_2...a_m$ en $y = b_1b_2...b_n$ dan is $xy = a_1a_2...a_mb_1b_2...b_n$. Concatenatie is associatief: $(xy)z = x(yz)$. De lege string gedraagt zich als een eenheidselement ten opzichte van de concatenatie omdat $\varepsilon x = x\varepsilon = x$ voor alle $x \in \Sigma^*$.

Als een string $z \in \Sigma^*$ van de vorm xy is met $x, y \in \Sigma^*$ dan heet x een *prefix* van z en heet y een *postfix* van z. Als bijvoorbeeld $z = 00110$ dan is ε een prefix van z evenals 0, 001, 0011 en z zelf. De postfixen van z zijn ε, 0, 10, 110, 0110 en opnieuw z zelf.

Als $x, z \in \Sigma^*$ met $z = wxy$ voor een paar $w, y \in \Sigma^*$ dan heet x een *substring* van z. De substrings van $z = 00110$ zijn dus ε, 0, 1, 00, 01, 10, 11, 001, 110, 0011, 0110 en z zelf.

Een (*formele*) *taal* L over een alfabet Σ wordt nu gedefinieerd als elke willekeurige deelverzameling van Σ^*. Als L_1 en L_2 twee van dergelijke talen zijn dan is hun *concatenatie* de taal $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$. Als bijvoorbeeld $L_1 = \{01,0\}$ en $L_2 = \{\varepsilon,0,10\}$ dan is $L_1L_2 = \{01,0,010,00,0110\}$. Ook deze (verzameling) concatenatie is associatief maar niet commutatief.

Voor elke taal L geldt dat $L\{\varepsilon\} = \{\varepsilon\}L = L$ en dus is de singletonverzameling $\{\varepsilon\}$ een eenheidselement voor de concatenatie van verzamelingen. De singletonverzameling met als element de lege string is iets geheel anders dan de lege verzameling ϕ. Deze laatste gedraagt zich als een nul-element ten opzichte van concatenatie van verzamelingen omdat $L\phi = \phi L = \phi$ voor elke taal L.

Het resultaat van concatenatie van een taal L met zichzelf LL wordt geschreven als L^2. Deze definitie wordt gegeneraliseerd naar $L^0 = \{\varepsilon\}$, $L^1 = L$ en $L^i = LL^{i-1} = L^{i-1}L$ voor $i \geqslant 2$. De *Kleene insluiting* van L, notatie L^* wordt vervolgens gedefinieerd als

$$\bigcup_{i=0}^{\infty} L^i .$$

L^+ is gedefinieerd als

$$\bigcup_{i=1}^{\infty} L^i$$

en dus is $L^* = L^+ \cup \{\varepsilon\}$. Σ^2 vertegenwoordigt alle strings uit Σ^* met lengte 2, Σ^3 vertegenwoordigt alle strings uit Σ^* met lengte 3, enzovoort. De verzameling strings over Σ met een lengte groter of gelijk één wordt aangegeven met Σ^+ en dus is

$$\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i \quad \text{en} \quad \Sigma^* = \Sigma^+ \cup \{\varepsilon\} = \bigcup_{i=0}^{\infty} \Sigma^i$$

Bij veel van onze berekeningen zullen we functies op strings toepassen en hierbij zullen de volgende functies in het bijzonder van pas blijken te komen:

$$kop(x) \colon \Sigma^* \rightarrow \Sigma^*$$

met als definitie

$$kop(x) = \begin{cases} \varepsilon & \text{als } x = \varepsilon \\ a & \text{als } x = ay, a \in \Sigma, y \in \Sigma^* \end{cases}$$

en evenzo

$$staart \colon \Sigma^* \rightarrow \Sigma^*$$

met als definitie

$$staart(x) = \begin{cases} \varepsilon & \text{als } x = \varepsilon \\ y & \text{als } x = ay, a \in \Sigma, y \in \Sigma^* \end{cases}$$

Elke $x \in \Sigma^*$ kan dus geschreven worden als de concatenatie van $kop(x)$ met $staart(x)$. Als bijvoorbeeld $x = 0111$ dan is $kop(x) = 0$ en $staart(x) = 111$. Voor de functie $staart_o staart$ schrijven we $staart^2$, voor $staart_o staart_o staart$ schrijven we $staart^3$, enzovoort. De eenheidsfunctie $staart^0$ definiëren we als $staart^0(x) = x$ voor alle $x \in \Sigma^*$. We kunnen $staart^i$ met $(i \geqslant 0)$ formeel recursief definiëren door

$$staart^i = \begin{cases} ident & als\ i = 0 \\ staart_o staart^{i-1} & als\ i \geqslant 1 \end{cases}$$

waarbij $ident(x) = x$ voor alle x.
Als dus $x = 0111$ dan is $staart^0(x) = 0111$, $staart(x) = 111$, $staart^2(x) = 11$, $staart^3(x) = 1$ en $staart^i(x) = \varepsilon$ voor $i \geqslant 4$.

Analoog aan *kop* en *staart* definiëren we de functies *voet* en *top*: $\Sigma^* \rightarrow \Sigma^*$ door

$$voet(x) = \begin{cases} \varepsilon & \text{als } x = \varepsilon \\ a & \text{als } x = ya, a \in \Sigma, y \in \Sigma^* \end{cases}$$

en

$$top(x) = \begin{cases} \varepsilon & \text{als } x = \varepsilon \\ y & \text{als } x = ya, a \in \Sigma, y \in \Sigma^* \end{cases}$$

Als x = 0111 dan is $voet(x)$ = 1 en $top(x)$ = 011.
We definiëren vervolgens

$$top^i = \begin{cases} ident & \text{als } x = 0 \\ top_o top^{i-1} & \text{als } i \geqslant 0 \end{cases}$$

Concatenatie kunnen we beschouwen als een functie $\Sigma^* \times \Sigma^* \to \Sigma^*$. Als dit voor de duidelijkheid nodig is zullen we $concat(x,y)$ schrijven in plaats van xy. Dus bijvoorbeeld $x = concat(kop(x), staart(x))$ of $x = concat(top(x), voet(x))$.

Stel Σ en Δ zijn twee alfabetten. Als nu een functie $f:\Sigma^* \to \Delta^*$ voldoet aan

$$f(\varepsilon) = \varepsilon$$

en aan $f(concat(x,x)) = concat(f(x), f(y))$ voor alle $x, y \in \Sigma^*$ dan noemen we f een *homomorfisme*.

Als we een homomorfisme $f:\Sigma^* \to \Delta^*$ definiëren dan hoeven we alleen $f(a)$ voor alle $a \in \Sigma$ te definiëren. Wegens de eigenschappen van een homomorfisme kunnen we dan $f(x)$ bepalen voor elke $x \in \Sigma^*$. Beschouw bijvoorbeeld het homomorfisme $linker: \{0,1\}^* \to \{0,1,\$\}^*$ gedefinieerd door $linker(0) = 0\$$ en $linker(1) = 1\$$. We weten dan dat $linker(\varepsilon) = \varepsilon$, $linker(0) = 0\$$, $linker(1) = 1\$$, $linker(00) = 0\$0\$$, $linker(01) = 0\$1\$$, $linker(10) = 1\$0\$$, $linker(11) = 1\$1\$$, $linker(000) = 0\$0\$0\$$, enzovoort.

In het vervolg zullen we veel te maken krijgen met functies $B^n \to B$ waarbij $B = \{0,1\}^*$. In hoofdstuk 5 zullen we nagaan welke functies uit deze klasse berekend kunnen worden met behulp van een Turingmachine. Dit geldt in ieder geval voor *kop*, *staart*, *voet*, *top* en *concat* die we hiervoor beschreven.

De functies *kop* en $staart: B \to B$ kunnen we als volgt definiëren:

$$\begin{aligned} kop(\varepsilon) &= \varepsilon \\ kop(0x) &= 0 \\ kop(1x) &= 1 \end{aligned}$$

en

$$\begin{aligned} staart(\varepsilon) &= \varepsilon \\ staart(0x) &= x \\ staart(1x) &= x. \end{aligned}$$

Om nu bijvoorbeeld $kop(y)$ te evalueren vergelijken we y met de strings ε, $0x$ en $1x$. Als $y = \varepsilon$ dan is $kop(y) = \varepsilon$; als y met een 0 begint dan is y van de vorm $0x$ voor een of andere string $x \in B$ en dus is $kop(y) = 0$; is y van de vorm $1x$ dan is $kop(y) = 1$.

De functies *voet* en *top* kunnen op een zelfde manier worden gedefinieerd:

$$voet(\varepsilon) = \varepsilon$$
$$voet(x0) = 0$$
$$voet(x1) = 1$$

en

$$top(\varepsilon) = \varepsilon$$
$$top(x0) = x$$
$$top(x1) = x$$

Willen we echter in de definities van de functies *voet* en *top* de parameters ε, $0x$ en $1x$ gebruiken dan moeten we ze recursief definiëren:

$$voet(\varepsilon) = \varepsilon$$
$$voet(0x) = als\ x = \varepsilon\ dan\ 0\ anders\ voet(x)$$
$$voet(1x) = als\ x = \varepsilon\ dan\ 1\ anders\ voet(x)$$

en

$$top(\varepsilon) = \varepsilon$$
$$top(0x) = als\ x = \varepsilon\ dan\ \varepsilon\ anders\ concat(0, top(x))$$
$$top(1x) = als\ x = \varepsilon\ dan\ \varepsilon\ anders\ concat(1, top(x))$$

Dit soort recursieve definities van functies, waarbij alle mogelijke waarden van de parameters worden aangegeven, zullen we in de volgende hoofdstukken nog regelmatig tegenkomen.

GERICHTE GRAFEN

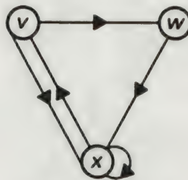
Een *gerichte graaf* (ook wel *digraaf* genoemd naar het Engelse 'directed graph') $G = (V, E)$ bestaat uit een eindige verzameling *vertices* (of *knopen*) en een relatie E op V. Een digraaf kan als volgt schematisch worden weergegeven: voor elke $v \in V$ tekenen we een knoop ⓥ en

als $(v,w) \in E$ dan verbinden we het knooppunt v met het knooppunt w door middel van een pijl zoals in Figuur 1.4.



Figuur 1.4

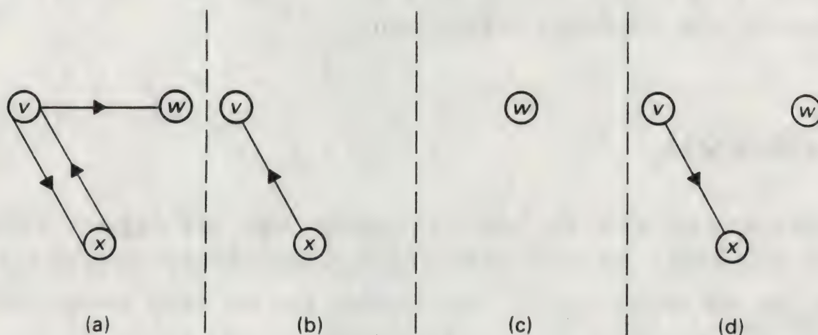
Figuur 1.5 geeft de digraaf $G_1 = (V_1, E_1)$ weer met $V_1 = \{v, w, x\}$ en $E_1 = \{(v,w), (v,x), (w,x), (x,v), (x,x)\}$.



Figuur 1.5

Als voor een bepaalde digraaf $G = (V,E)$ geldt dat $(v,w) \in E$ dan noemen we dit een *kant* of *verbinding* van v naar w in G. Als er een rij knopen bestaat $v = v_0, v_1, \ldots, v_n = w$ met $n \geqslant 0$ zo dat $(v_i, v_{i+1}) \in E$ voor $i = 1,2,\ldots,n$ dan is er een *gericht pad* ter lengte n van v naar w in G. Als er een pad tussen v en w bestaat dan betekent dit dat ofwel $v = w$, ofwel w is vanuit v bereikbaar door in het diagram een rij gerichte kanten te volgen.

Als $G = (V,E)$ een digraaf is, dan heet elke digraaf (V', E') met $V' \subseteq V$ $E' \subseteq E$ een *subgraaf* van G. In Figuur 1.6 zijn vier gerichte subgrafen van de digraaf G_1 uit Figuur 1.5 weergegeven.



Figuur 1.6

Twee gerichte subgrafen heten disjunct als zij geen knooppunten gemeen hebben. In Figuur 1.6 zijn (b) en (c) de enige twee disjuncte gerichte subgrafen.

Als $V = \{v_1, v_2, \ldots, v_n\}$ en $G = (V,E)$ is een digraaf dan kunnen we G ook weergeven met behulp van een $n \times n$ Boole'se matrix M_G die de *nabijheidsmatrix* (Engels: adjacency matrix) wordt genoemd. M_G wordt gedefinieerd door

$$M_G[i,j] = \begin{cases} \text{T} & \text{als } (v_i, v_i) \in E \\ \text{F} & \text{anders} \end{cases}$$

(T staat voor True of Waar en F voor False of Onwaar.)

Als we bijvoorbeeld de elementen van V_1 alfabetisch rangschikken dan kan de digraaf uit Figuur 1.5 ook voor de volgende Boole'se matrix worden weergegeven:

$$\begin{bmatrix} \text{F} & \text{T} & \text{T} \\ \text{F} & \text{F} & \text{T} \\ \text{T} & \text{F} & \text{T} \end{bmatrix}$$

In computerprogramma's worden digrafen vaak op deze manier weergegeven. Als het gaat om een grote verzameling knooppunten dan wordt de matrix erg groot en kan een van de vele methoden worden gebruikt voor het opslaan van zogenaamde 'schaarse' matrices, (zie Oefening 1.21).

Als $G = (V,E)$ een digraaf is en $v \in V$ dan noemen we het aantal uitgaande kanten vanuit v de *uitgraad* van v. Het aantal inkomende kanten in v heet de *ingraad*. Ingraad en uitgraad zijn functies $V \to Z^+$ (Z^+ is de verzameling van niet-negatieve gehele getallen). De functies worden als volgt gedefinieerd:

$$uitgraad(v) = \#\{(v,w) \mid (v,w) \in E\}$$
$$ingraad(v) = \#\{(w,v) \mid (w,v) \in E\}$$

Vervolgens kunnen we de functie $graad: V \to Z^+$ definiëren door

$$graad(v) = uitgraad(v) + ingraad(v)$$

Omdat elke kant zijn oorsprong heeft in een knoop en ook leidt naar een knoop geldt de volgende stelling:

Stelling 1.3

Voor elke digraaf $G = (V, E)$ geldt

(1) $\sum_{v \in V} uitgraad(v) = \sum_{v \in V} ingraad(v) = \#(E)$

(2) $\sum_{v \in V} graad(v) = 2\#(E)$

Voor de digraaf G_1 kunnen de functies *uitgraad*, *ingraad* en *graad* als volgt in een tabel worden weergegeven:

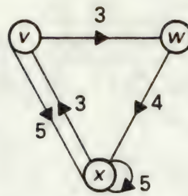
	uitgraad	*ingraad*	*graad*
v	2	1	3
w	1	1	2
x	2	3	5

Bij veel toepassingen is het zinvol de kanten in een digraaf te labelen met labels geselecteerd uit een verzameling labels L. Een *gelabelde digraaf* is dan een digraaf $G = (V,E)$ te zamen met een labelfunctie $l:E \to L$. Een gelabelde digraaf wordt schematisch weergegeven zoals een ongelabelde digraaf, maar met vermelding van het label $l(e)$ bij elke kant e. De digraaf G kan bijvoorbeeld worden gelabeld met behulp van een functie $l:E \to Z^+$ met als definitie

$$l((v,w)) = uitgraad(v) + ingraad(w) \quad \text{voor alle} \quad (v,w) \in E.$$

Het resultaat is schematisch weergegeven in Figuur 1.7.

Voor het representeren van een gelabelde digraaf $G = (V,E)$ met een labelfunctie $l:E \to L$ binnen de computer, gebruiken we meestal een *labelmatrix*. Om te beginnen ordenen we de elementen van V bijvoorbeeld als $v_1, v_2, \ldots, v_n$. De i-de rij en de j-de kolom bepalen dan het



Figuur 1.7

label dat hoort bij de kant (v_i, v_j) als deze tenminste bestaat. Anders bevat het element een speciaal teken $\omega \notin L$ om aan te geven dat v_i en v_j niet direct verbonden zijn. De labelmatrix M_G kan formeel worden gedefinieerd als een $n \times n$ matrix met elementen uit $L \cup \{\omega\}$ met als waarden

$$M_G[i,j] = \begin{cases} \omega & \text{als } (v_i, v_j) \notin L \\ l((v_i, v_j)) & \text{anders} \end{cases}$$

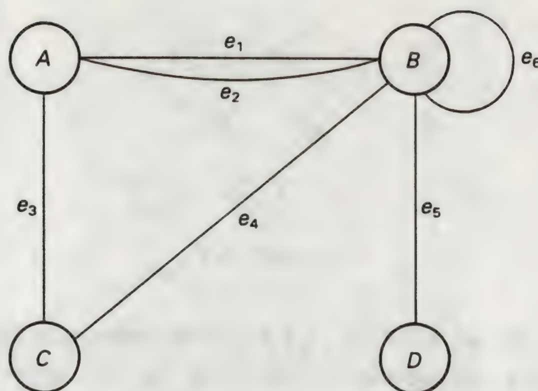
Voor het weergeven van de gelabelde digraaf uit Figuur 1.7 gebruiken we $\omega = -1$ en we ordenen de knooppunten alfabetisch. De labelmatrix wordt dan

$$\begin{bmatrix} -1 & 3 & 5 \\ -1 & -1 & 4 \\ 3 & -1 & 5 \end{bmatrix}$$

GRAFEN

Binnen gerichte grafen heeft elke kant of verbinding tussen twee knopen een richting: de kant (v, w) loopt van v naar w. Ook geldt omdat de kanten zijn gedefinieerd als een deelverzameling van $V \times V$ dat er hoogstens één verbinding kan lopen tussen elk knooppunt v en elk ander knooppunt w. Deze eigenschappen zijn in veel gevallen nuttig, maar soms vormen zij een te grote beperking.

We definiëren daarom een *graaf* $G = (V, E)$ als een structuur die bestaat uit een eindige verzameling *knooppunten* V en een verzameling *verbindingen* of *kanten* E. Elke kant $e \in E$ verbindt twee knooppunten die we de eindpunten van e zullen noemen. Een graaf wordt op bijna dezelfde manier als een digraaf schematisch weergegeven. In figuur 1.8



Figuur 1.8

hebben we een graaf getekend met vier knooppunten A, B, C en D en zes kanten $e_1, e_2, \ldots, e_6$. Twee kanten (e_1 en e_2) verbinden A en B en bij de kant e_6 zijn de twee eindpunten gelijk. Als een kant e de knopen v_1 en v_2 als eindpunten heeft dan geven we die kant ook wel door $v_1 \overset{e}{-} v_2$. Een kant met identieke eindpunten noemen we een *lus* en twee kanten die dezelfde eindpunten hebben noemen we *parallel*. Het voorbeeld in Figuur 1.8 heeft twee parallelle kanten en één lus.

De graad van knoop $v \in V$, $graad(v)$ is het aantal malen dat v een eindpunt is voor een kant. Zo is $graad(A) = 3$, $graad(B) = 6$, $graad(C) = 2$ en $graad(D) = 1$. Elk knooppunt v met $graad(v) = 0$ noemen we *geïsoleerd*. Elke kant draagt één eenheid bij aan de graad van zijn eindpunten. Overeenkomstig met Stelling 1.3 geldt dus voor elke graaf $G = (V,E)$

$$\sum_{v \in V} graad(v) = 2\#(E)$$

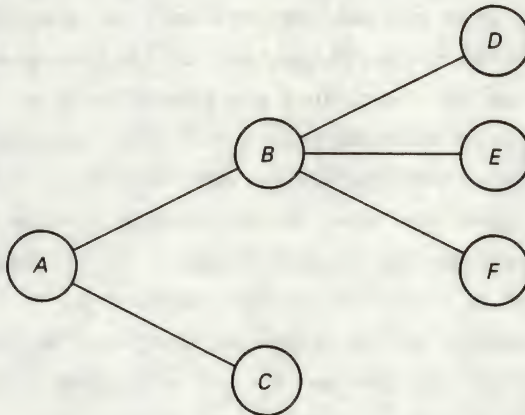
Een *complete graaf* op V bezit precies één kant tussen elk paar verschillende knopen $v, w \in V$. Als K_n een complete graaf is op n knopen, dan heeft elke knoop graad $n-1$ en heeft de graaf totaal $n(n-1)/2$ kanten. K_n heeft geen lussen en geen parallelle kanten.

Een *pad* in een graaf G is een rij kanten $e_1, e_2, \ldots$ zodanig dat (1) e_i en e_{i+1} een gemeenschappelijk eindpunt hebben en (2) als e_i niet het eerste en niet het laatste eindpunt is dan heeft e_i zijn ene eindpunt gemeen met e_{i-1} en zijn andere met e_{i+1}. In Figuur 1.8 is $e_1e_2e_3$ een pad evenals $e_1e_6e_2e_3$, maar bijvoorbeeld $e_5e_3e_4$ is geen pad.

Als $e_1e_2...e_k$ een pad is en k heeft een eindige waarde, dan zeggen we dat het pad *lengte h* heeft. We kunnen zo'n pad als volgt beschrijven: $v_0 \overset{e_1}{—} v_1 \overset{e_2}{—} \ldots v_{k-1} \overset{e_k}{—} v_k$. Het pad *begint* bij v_0 en *eindigt* bij v_k. Het pad *verbindt* v_0 en v_k. In het voorbeeld is $e_1e_2e_3$ een pad dat A en C verbindt.

Een *circuit* is een pad met het eindpunt gelijk aan het beginpunt. Een pad heet *enkelvoudig* als geen knooppunt er meer dan één keer in voorkomt. Een circuit heet enkelvoudig als alleen het begin/eindpunt er twee keer en niet meer dan twee keer in voorkomt. Het voorbeeld in Figuur 1.8 bevat $e_3e_1e_5$ en e_4e_5 als enkelvoudige paden tussen C en D. De paden $e_4e_2e_1e_5$ en $e_3e_2e_6e_5$ zijn niet enkelvoudig. Verder is $e_4e_2e_3$ een enkelvoudig circuit, maar $e_3e_1e_2e_4$ is dat niet. Een graaf zonder circuits heet *circuit-vrij*.

Een graaf G heet *verbonden* als er tussen elke twee knooppunten u en v een pad in G bestaat dat deze knooppunten verbindt. Een verbonden, circuit-vrije graaf heet een *boom*. De graaf in Figuur 1.9 is een boom.



Figuur 1.9

Stelling 1.4

Als $G = (V,E)$ een graaf is dan zijn de volgende eigenschappen equivalent:

(1) G is een boom

(2) G is circuit-vrij, maar het toevoegen van een kant leidt tot een circuit

(3) G bevat geen lussen en tussen elk paar $v,w \in V$ bestaat er een uniek pad

(4) G is verbonden, maar het verwijderen van een kant heft deze eigenschap op

(5) G is circuit-vrij en heeft $\#(V) - 1$ kanten

(6) G is verbonden en heeft $\#(V) - 1$ kanten

Bewijs

(1) $\Rightarrow$ (2): $G = (V,E)$ is een boom en dus verbonden en circuit-vrij. Stel $e \in E$ wordt als verbinding tussen v en w toegevoegd. Veronderstel dat $v \neq w$ omdat anders een lus zou voorkomen. Omdat G verbonden is was er al een pad tussen v en w; het toevoegen van e moet dus leiden tot een circuit.

(2) $\Rightarrow$ (3): G is circuit-vrij en bevat dus geen lussen. Beschouw nu $v,w \in V$. Als er meer dan één pad zou zijn tussen v en w dan bevat G een circuit. Als er geen pad zou zijn tussen v en w dan leidt dit tot een tegenspraak omdat het toevoegen van de kant $v - w$ een circuit moet vormen. Dus moet er een uniek pad zijn tussen v en w.

(3) $\Rightarrow$ (4): Omdat er voor elk paar $v,w \in V$ een pad tussen v en w loopt is G verbonden. Veronderstel nu dat de kant $v - w$ wordt verwijderd. Er kan dan geen pad meer bestaan tussen v en w omdat de verwijderde kant het unieke pad vormde tussen v en w. Door het verwijderen van een kant is G dus niet langer verbonden.

(4) $\Rightarrow$ (1): We moeten aantonen dat G circuit-vrij is. Dit is waar, want als G een circuit had dan zou na het verwijderen van een willekeurige kant uit dit circuit G toch nog verbonden zijn.

Nu we de equivalentie van de eigenschappen (1), (2), (3) en (4) hebben aangetoond, kunnen we deze eigenschappen gebruiken om aan te tonen dat de eigenschappen (5) en (6) eveneens equivalent zijn en slechts gelden als G een boom is. We gebruiken inductie naar $\#(V)$. Allereerst tonen we aan dat elke boom $G = (V,E)$, $\#(V) - 1$ kanten of takken moet hebben. Voor $\#(V) = 1$ is dit zeker waar, dus veronderstellen we $\#(V) \geqslant 2$. Beschouw het geval $\#(V) = n$ en veronderstel dat de te bewijzen bewering waar is voor alle bomen met minder dan n

knooppunten. Omdat G een boom is, is G circuit-vrij en verbonden. Als we een tak uit G verwijderen dan wordt G gesplitst in twee delen $G_1 = (V_1, E_1)$ en $G_2 = (V_2, E_2)$ die elk op zich verbonden en circuit-vrij zijn. G_1 en G_2 zijn dus beiden bomen en hebben dus op grond van de inductieveronderonderstelling respectievelijk $\#(V_1) - 1$ en $\#(V_2) - 1$ takken. Totaal bevat G dan $(\#(V_1)-1) + (\#(V_2)-1) + 1 = \#(V_1 \cup V_2) - 1 = \#(V) - 1$ takken en hiermee is het bewijs door inductie geleverd. Dus geldt (1) $\Rightarrow$ (5) en (1) $\Rightarrow$ (6).

De volgende stap in het bewijs bestaat uit het aantonen dat uit het feit dat G verbonden is en $\#(V) - 1$ kanten heeft volgt dat G circuit-vrij is en omgekeerd dat als G circuit-vrij is en $\#(V) - 1$ kanten heeft dat G dan noodzakelijkerwijs verbonden is. Deze beweringen zijn zeker waar voor $\#(V) = 1$, dus veronderstellen we dat $\#(V) \geqslant 2$.

Veronderstel dat G verbonden is en $\#(V) - 1$ kanten heeft. Als G circuits bevat dan kunnen we kanten verwijderen terwijl G toch verbonden blijft. Als G geen circuits meer bevat en dus geen kanten meer kunnen verwijderen dan hebben we een verbonden graaf zonder circuits en dus een boom. We hebben echter zo juist bewezen dat een boom $\#(V)-1$ kanten of takken heeft en hieruit moeten we concluderen dat we geen kanten uit G kunnen verwijderen, zo dat G toch verbonden blijft. G is dus circuit-vrij.

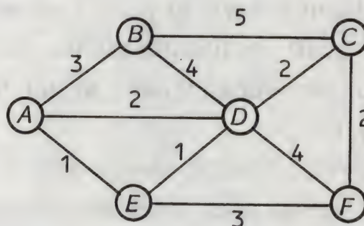
Laten we nu veronderstellen dat G circuit-vrij is en $\#(V)-1$ kanten bevat. Kies nu een willekeurige kant $e \in E$ en verleng deze tot een pad door steeds een kant aan een eindpunt toe te voegen. Omdat G circuit-vrij is kan dit toevoegen maar een eindig aantal malen gebeuren en krijgen we een pad met twee eindpunten, ieder met graad 1. We verwijderen nu één van die eindpunten en de bijbehorende kant uit G. Het resultaat is een graaf $G_1 = (V_1, E_1)$ met $\#(V_1) = \#(V)-1$ en $\#(E_1) = \#(E)-1$. Ook deze graaf is circuit-vrij en met behulp van inductie kunnen we aantonen dat hij verbonden moet zijn. Dus moet G ook verbonden zijn. Hiermee hebben we bewezen dat (5) $\Leftrightarrow$ (6).

Omdat per definitie geldt dat (5) $\Rightarrow$ (1) en (6) $\Rightarrow$ 1 is hiermee de hele stelling bewezen.

Een graaf $G^1 = (V^1, E^1)$ is een *subgraaf* van een graaf $G = (V, E)$ als $V^1 \subseteq V$ en $E^1 \subseteq E$. Als G^1 alle knooppunten van G bevat dan heet G^1

een *opspannende subgraaf* van G. Als daarbij G^1 ook een boom is dan heet G^1 een *opspannende boom* van G.

Stel we hebben een graaf $G = (V,E)$ en een kostenfunctie $c : E \to \mathbb{R}^+$ die een positief reëel getal toekent aan elke kant in G. We kunnen deze situatie in een diagram weergeven door elke kant in de graaf te labelen met zijn bijbehorende kosten. In Figuur 1.10 wordt daarvan een voorbeeld gegeven. We definiëren de kosten van zo'n graaf als $c(G) = \sum_{e \in E} c(e)$. De kosten van de graaf in figuur 1.10 bedragen dus 27. Een veel voorkomend probleem is het bepalen van een verbonden opspannende subgraaf G^1 van een dergelijke graaf, zodanig dat $c(G^1)$ minimaal is. Duidelijk is dat G^1 circuit-vrij moet zijn omdat we anders de kosten zouden kunnen verminderen door een kant te verwijderen terwijl de subgraaf toch verbonden blijft. G^1 is dus een boom en heet de *minimale opspannende boom* (Engels: Minimal Spanning Tree (MST)) van G. Er bestaan verschillende algoritmen voor het construeren van een MST. Het volgende algoritme is ontworpen door Kruskal (1956).



Figuur 1.10

Zij $G = (V, E)$ de graaf. We kunnen ervan uitgaan dat G geen parallelle kanten bevat omdat we anders deze kanten op de goedkoopste na zouden kunnen verwijderen. Het algoritme ter bepaling van de in de boom op te nemen kanten werkt nu als volgt:

begin $F \leftarrow 0$;
zolang $\#(F) < \#(V)-1$ *doe*
selecteer $e \in E \setminus F$ *zo dat*
(1) e geen circuit vormt samen met al in F opgenomen kanten
(2) $c(e)$ zo klein mogelijk is terwijl aan (1) voldaan is
$F := F \cup \{e\}$
eindezolang;
output(F)
einde

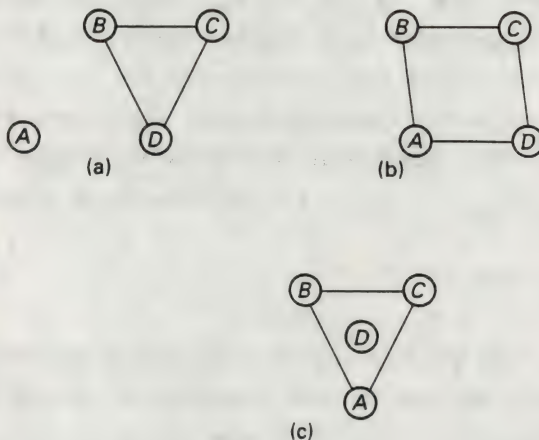
Uit de graaf in Figuur 1.10 worden door dit algoritme de kanten A-E, E-D, D-C, C-F en A-B geselecteerd. Op grond van stelling 1.4 leidt Kruskal's algoritme tot een opspannende boom. Het is niet moeilijk aan te tonen dat de bijbehorende kosten minimaal zijn. Veronderstel dat het algoritme de kanten $e_1, e_2, \ldots, e_{n-1}$ selecteert met $\#(V) = n$, terwijl $c(e_1) \leqslant c(e_2) \leqslant \ldots \leqslant c(e_{n-1})$. Stel verder dat T de goedkoopst mogelijke opspannende boom is en dat T de verzameling kanten E bevat. Neem nu aan dat $E \neq \{e_1, e_2, \ldots, e_{n-1}\}$. Dit betekent dat er minstens één index k moet bestaan met $e_k \notin E$. E is dan bijvoorbeeld gelijk aan $\{e_1, e_2, \ldots, e_{k-1}, f_k, f_{k+1}, \ldots, f_{n-1}\}$ en wegens de keuze die wordt gedaan in Kruskal's algoritme geldt $c(e_k) \leqslant \min\{c(f_j) \mid k \leqslant j \leqslant n\}$. $E \cup \{e_k\}$ moet noodzakelijkerwijs een circuit bevatten waarin e_k voorkomt. Dit circuit bevat ook minstens één f_j met $k \leqslant j < n$. Vervang nu f_j door e_k, dat wil zeggen beschouw $(E \setminus \{f_j\}) \cup \{e_k\}$. Deze verzameling kanten vormt weer een boom, maar heeft kosten $\leqslant c(T)$. Omdat $c(T)$ al minimaal was moeten de kosten nog steeds gelijk aan $c(T)$ zijn en dus moet $c(f_j) = c(e_k)$. Deze zelfde redenering geldt voor alle elementen $f_k, f_{k+1}, \ldots, f_{n-1}$. Zij kunnen stuk voor stuk vervangen worden door elementen uit $e_k, e_{k+1}, \ldots, e_{n-1}$ en dus kunnen we concluderen dat

$$c(T) = \sum_{i=1}^{n-1} c(e_i)$$

Het algoritme van Kruskal voor het genereren van een minimale opspannende boom is één van de vele voorbeelden van algoritmen voor grafen. Goede overzichten van dergelijke algoritmen staan in *Graphs Algorithms* (Even, 1979) en in *The Design and Analysis of Computer Algorithms* (Aho et al., 1974).

Bij veel toepassingen zijn de namen van de knooppunten in een graaf eigenlijk niet interessant. In figuur 1.11 kunnen we de grafen (a) en (b) verschillend noemen, evenals de grafen (b) en (c), maar (a) en (c) zijn in feite niet verschillend. Als we in (a) de knooppunten A,B,C,D herlabelen als D,B,C,A dan zijn beide grafen identiek. Twee grafen die alleen in de namen van hun knooppunten verschillen heten *isomorf*. De formele definitie luidt: twee grafen $G_1 = (V_1,E_1)$ en $G_2 = (V_2,E_2)$ heten isomorf desd als er bijecties $f:V_1 \rightarrow V_2$ en $g:E_1 \rightarrow E_2$ bestaan, zodanig dat voor elke kant $u \overset{e}{—} v$ in G_1 er een kant $f(u) \overset{g(e)}{——} f(v)$ in G_2 bestaat. Isomorfisme is een equivalentie-relatie op de verzameling van alle grafen.

Het is niet altijd even gemakkelijk vast te stellen of twee grafen isomorf zijn. Twee isomorfe grafen moeten in ieder geval evenveel knopen en kanten hebben. Verder moet voor elke bijectie $f:V_1 \rightarrow V_2$ die wordt gebruikt om het isomorfisme vast te stellen, gelden dat $graad(v) = graad(f(v))$ voor alle $v \in V$. Er bestaan nog andere soortgelijke eigenschappen waarmee het aantal te beschouwen bijecties kan worden beperkt. In het slechtste geval echter kan het toch voorkomen als V_1 en V_2 elk n knopen hebben, dat we alle $n!$ mogelijke bijecties moeten bekijken om het gezochte isomorfisme te bewijzen. Helaas is er op dit moment, ondanks veel onderzoek, geen enkel snel algoritme voor het vaststellen van isomorfisme van grafen.



Figuur 1.11

GÖDEL NUMMERING

Bij elke aftelbare verzameling A kunnen we een één-eenduidige functie vaststellen van A op de verzameling van de natuurlijke getallen N. Elk element $x \in A$ kan dus worden gecodeerd als een uniek getal $g(x)$. Het zou nu kunnen voorkomen dat we een codering g hebben ontworpen met een waardebereik dat een deelverzameling is van N. In zo'n geval willen we voor elke $i \in N$ kunnen vaststellen of i voorkomt in het waardebereik van g en als dat zo is dan willen we $g^{-1}(i)$ kunnen bepalen. Een ordening van de elementen van A kunnen de definiëren door af te spreken dat $x \in A$ voorafgaat aan $y \in A$ desd als $g(x) < g(y)$. De eerste n elementen van A kunnen we dan met behulp van het volgende algoritme bepalen.

```
i:= 1; j:= 1;
zolang j ≤ n doe
    als i in het waardebereik van g voorkomt
        dan drukaf (g⁻¹(i)); j:= i + 1
    eindeals;
    i:= i+1
eindezolang
```

$A = \{0,1\}^*$ kunnen we als volgt coderen. Voor elke x $\{0,1\}^*$ definiëren we de gehele waarde $g(x)$ door:

$$g(x) = \begin{cases} 1 & \text{desd } x = \varepsilon \\ p_1^{a_1} \times p_2^{a_2} \times \ldots \times p_n^{a_n} & \text{als } x = b_1 b_2 \ldots b_n \text{ met } b_i \in \{0,1\} \end{cases}$$

waarbij $1 \leq i \leq n$, p_i het i-de priemgetal voorstelt en $a_i = b_i + 1$.

Als bijvoorbeeld x de string 1100 voorstelt dan is $g(x) = 2^2 \times 3^2 \times 5 \times 7 = 1260$.

De functie g is een voorbeeld van een Gödel nummering van de verzameling $\{0,1\}^*$. Een *Gödel nummering* van een verzameling is een toewijzing van natuurlijke getallen aan de elementen van de verzameling, die voldoet aan de volgende voorwaarden:

(1) Bij verschillende elementen horen verschillende Gödelgetallen (g is dus een één op één functie);

(2) Het Gödelgetal van elk element kan met behulp van een algoritme worden berekend (de functie g is dus effectief berekenbaar).

(3) Van ieder natuurlijk getal kan worden vastgesteld of het een Gödelgetal van een element uit de verzameling is en als dit het geval is dan kan dit element ook worden bepaald (ook g^{-1} is effectief berekenbaar).

In dit boek zullen we steeds Gödelnummering gebruiken met de volgende eigenschap: als $m < n$ en $g(x)$ is deelbaar door p_n dan is $g(x)$ ook deelbaar door p_m. Het algoritme voor het decoderen van Gödelgetallen is dan vrij eenvoudig.

In bovenstaand voorbeeld is de functie g inderdaad een één op één functie omdat elk geheel getal maar op één manier in priemfactoren ontbonden kan worden. Het waardebereik van g bevat het getal 1 en verder alle gehele getallen die voor een of andere n een ontbinding in priemfactoren van de vorm $p_1^{a_1} \times p_2^{a_2} \times \ldots \times p_n^{a_n}$ hebben, met $a_i \in \{1,2\}$ voor $1 \leqslant i \leqslant n$. De waarden van g kleiner of gelijk 30 zijn dan 1,2,4,6,12,18 en 30 en daarbij behoren de strings ε,0,1,00,10,01,000. Merk op dat de ordening die we verkrijgen met de Gödelnummering niet gelijk is aan de eerder beschreven lexicografische ordening.

De Gödelnummering is een geschikte coderingstechniek en we illustreren dit met nog een voorbeeld. Zij Γ de verzameling van alle grafen met ongelabelde knooppunten. In Figuur 1.12(a) is een element van getekend met vier knopen en vijf kanten. Γ bevat precies één graaf voor elke equivalentieklasse onder isomorfisme. We kunnen bewijzen dat Γ aftelbaar oneindig is door aan elke graaf $G \in \Gamma$ een Gödelgetal $g(G)$ toe te kennen. Allereerst kunnen we de knooppunten van G labelen met $p_1, p_2, \ldots, p_n$ en een mogelijk Gödelgetal wordt dan bepaald door

$$p_1^{k_1} \times p_2^{k_2} \times \ldots \times p_n^{k_n}$$

waarbij k_i voor $1 \leqslant i \leqslant n$ wordt berekend met behulp van het volgende algoritme:

k_i:= 1;
voor j van 1 *tot n doe*
  e:= aantal kanten tussen p_i en p_j;
  $k_i := k_i \times p_j^e$
eindevoor

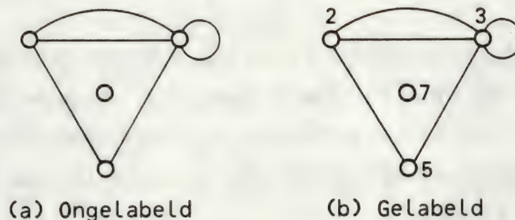
Dus $k_i = 1$ geldt desd als de knoop met label p_i geïsoleerd is.

Als we de graaf in figuur 1.12(a) labelen zoals in figuur 1.12(b) dan is het erbij behorende getal:

$$2^{3^2 \cdot 5} \times 3^{2^2 \cdot 3 \cdot 5} \times 5^{2 \cdot 3} \times 7 = 2^{45} \times 3^{60} \times 5^6 \times 7$$

Behoudens isomorfismen bestaat er maar één graaf met een bepaald getal als we de hierboven beschreven constructie gebruiken. Elke graaf kan echter wel op $n!$ verschillende manieren worden gelabeld als hij n knooppunten heeft, afhankelijk van de wijze waarop we de priemgetallen $p_1, p_2, \ldots, p_n$ aan de knooppunten toewijzen. Om de Gödelnummering toch uniek te maken definiëren we $g(G)$ als de kleinste van de $n!$ mogelijke uitkomsten. Gemakkelijk kan worden gecontroleerd dat g inderdaad een één op één functie is.

Gödelnummering is een handige coderingstechniek die het mogelijk maakt verschillende discrete structuren zoals grafen voor te stellen met behulp van gehele getallen. In het vervolg van dit boek zullen we daarom onze aandacht vooral richten op functies gedefinieerd over de natuurlijke getallen. We zullen daarbij gebruik maken van de binaire of tweetallige notatie van die getallen door middel van een string in $\{0,1\}^*$ en ons onderzoek zal daarom ook betrekking hebben op berekenbare functies over de verzameling $B = \{0,1\}^*$.



Figuur 1.12

OEFENINGEN

1. Als het beschouwde universum de verzameling positieve gehele getallen kleiner dan 20 is, som dan de elementen op van de volgende verzamelingen:

   (a) $\{x \mid x + 2 < 10\} \cup \{x \mid x \text{ is priem}\}$,

(b) $\{x \mid x = x^2\}$,

(c) $\{x \mid 2x = 1\}$,

(d) $\{x \mid 3x < 20\} \backslash \{x \mid x \text{ is even}\}$,

2. Zij A een eindige verzameling. Toon aan dat $\#(2^A) = 2^{\#(A)}$.

3. Laat zien dat Q, de verzameling van de rationale getallen, aftelbaar oneindig is.

4. Laat zien dat als Σ een eindig alfabet is, dat dan Σ^* aftelbaar oneindig is.

5. Als $R \subseteq A \times N$ een relatie is dan is de inverse R^{-1} een relatie in $B \times A$ gedefinieerd door

$$R^{-1} = \{(b,a) \mid (a,b) \in R\}$$

(a) Bewijs dat $(R^{-1})^{-1} = R$;

(b) Aan welke voorwaarden moet R voldoen opdat R^{-1} een totale functie is?

Als $A = B$ en R dus een relatie op A is, toon dan aan dat

(c) R is reflexief desd als R^{-1} reflexief is;

(d) R is symmetrisch desd als R^{-1} symmetrisch is;

(e) R is transitief desd als R^{-1} transitief is.

6. Toon aan dat de equivalentieklassen die worden gedefinieerd door een equivalentierelatie R op een verzameling A, deze verzameling partitioneren in een aantal disjuncte niet-lege verzamelingen. [Hint: $\bar{a}$ is niet-leeg omdat $a \in \bar{a}$; beschouw nu $\bar{a}$ en $\bar{b}$ en laat zien dat als $\bar{a} \cap \bar{b} \neq 0$ dan $\bar{a} = \bar{b}$.]

7. Laat zien dat als R_1 en R_2 relaties op A zijn dan geldt

(a) R_1 is reflexief impliceert dat $R_1 \cup R_2$ is reflexief;

(b) R_1 en R_2 zijn reflexief impliceert dat $R_1 \cap R_2$ is reflexief.

Blijven deze beweringen waar als we 'reflexief' overal vervangen door 'symmetrisch'? En hoe zit het bij vervanging door 'transitief' en bij vervanging door 'antisymmetrisch'?

8. Zij R een relatie in $A \times B$ en S een relatie in $B \times C$. De *compositie* S_0R van deze twee relaties is gedefinieerd als de relatie in $A \times C$ met

$$S_o R = \{(a,c) \text{ er bestaat een } b \in B \text{ met } (a,b) \in R \text{ en } (b,c) \in S\}$$

(a) Bewijs dat compositie een associatieve operatie is.

(b) Als S en S beide totale surjectieve functies zijn, bewijs dan dat $S_o R$ ook een totale functie van A op C is.

9. Laat zien hoe een relatie $R \subseteq \mathbb{R} \times \mathbb{R}$ kan worden weergegeven door middel van een verzameling punten in het Cartesische vlak. Als de lijn tussen de punten (x_1, y_1) en (x_2, y_2) een deelverzameling van is voor alle punten $(x_1, y_1) \in R$ en $(x_2, y_2) \in R$ dan heet R een *convexe verzameling*. Teken de volgende relaties in een Cartesisch coördinatenstelsel en geef aan welke een convexe verzameling zijn.

(a) $R = \{(x,y) \mid x+y \geqslant 2, x \geqslant 0, y \geqslant 0\}$

(b) $S = \{(x,y) \mid y \leqslant 2x+4, y \leqslant 4-2x, x^2+y^2 \geqslant 4, y \geqslant 0\}$

10. Welke van de volgende functies $\mathbb{R} \to \mathbb{R}$ zijn surjectief en welke zijn injectief of één op één? Geef vervolgens aan welke functies dus bijectief zijn.

(a) $cuub(x) = x^3$

(b) $mod(x) = \begin{cases} x & \text{als } x \geqslant 0 \\ -x & \text{als } x < 0 \end{cases}$

(c) $sin(x)$

(d) $exp(x) = e^x$

(e) $rec(x) = \begin{cases} 1 & \text{als } x = 0 \\ 1/x & \text{als } x \neq 0 \end{cases}$

11. Definieer, gebruikmakend van de definities uit opgave 10, zo eenvoudig mogelijk $exp_o rec$ en $rec_o cuub_o mod$.

12. Laat zien dat $f: A \to B$ surjectief moet zijn, wil $f^{-1}: B \to A$ een totale functie zijn en dat f^{-1} geen functie is als f niet injectief is. Bewijs vervolgens dat f een bijectie is dan en slechts dan als f^{-1} een totale functie is. Bepaal de inverse functies van de bijecties uit opgave 10.

13. Bewijs met inductie dat $3^{2n+1} + 4^{2n+1}$ deelbaar is door 7 voor alle $n \in N$.

14. Bewijs met inductie dat $S_n = 1^2 + 2^2 + \ldots + n^2$ gelijk is aan $n(n+1)(2n+1)/6$ voor alle $n \in N$.
Zij $C_n = 1^3 + 2^3 + \ldots + n^3$. Bereken C_1, C_2 en C_3 en maak een veronderstelling over een formule voor C_n. Bewijs de juistheid van de veronderstelde formule vervolgens met inductie. [Hint: probeer een verband te vinden tussen C_n en $1+2+\ldots+n = n(n+1)/2$.]

15. Bekijk onderstaande recursieve functiedefinities. Probeer een eenvoudige formule voor elke functie te vinden en bewijs die met inductie.

(a) $sq: N \to N$ met $sq(x)$ = *als* $x = 1$ *dan* 1
anders $sq(x-1) + 2x - 1$

(b) $mult: N \times N \to N$ met $mult(x,y)$ = *als* $x = 1$ *dan* y
anders $mult(x-1,y) + y$

(c) $macht: N \times N \to N$ met $macht(x,y)$ = *als* $x = 1$ *dan* 1
anders $mult(macht(x,y-1),x)$

16. Toon aan dat het aantal onderscheidbare partiële functies $N \to N$ niet aftelbaar is.

17. Z^+ is de verzameling van niet-negatieve gehele getallen. Toon aan dat $lengte: \Sigma^* \to Z^+$ gedefinieerd door

$$lengte(x) = \begin{cases} 0 & \text{als } x = \varepsilon \\ 1 + lengte(y) & \text{als } x = ay, a \in \Sigma, y \in \Sigma^* \end{cases}$$

voldoet aan $lengte(x) = |x|$

Gebruik deze recursieve definitie van de functie lengte om te bewijzen dat $lengte(xy) = lengte(x) + lengte(y)$ voor alle $x,y \in \Sigma^*$.

18. Formuleer een recursieve definitie van de functie $rev: \Sigma^* \to \Sigma^*$ zodanig dat als $x = a_1a_2\ldots a_n$ met $a_i \in \Sigma$ voor $i = 1,2,\ldots,n$ dan is $rev(x) = a_na_{n-1}\ldots a_1$.
Bewijs met behulp van de recursieve definities van lengte en rev dat $lengte(rev(x)) = lengte(x)$ voor alle strings $x \in \Sigma^*$.

19. Als $L_1 = \{a,bc,a\}$ en $L_2 = \{\varepsilon,bcc,a\}$ bepaal dan $L_1^2L_2 \setminus L_2^2L_1$.

20. Maak gebruik van de definitie van lengte uit opgave 17 om te bewijzen dat $staart^i(x) = \varepsilon$ desd als $top^i(x) = \varepsilon$ desd als $lengte(x) \leqslant i$.

21. M heet een *schaarse matrix* als de meerderheid van haar elementen gelijk is. In dat geval kan M worden voorgesteld door middel van een lijst van tripels die de onderling wel verschillende elementen representeert. Elk tripel bestaat uit de rij- en de kolomindex en de waarde van een element. Laat zien dat als M een $m \times n$ matrix is met geheeltallige elementen, dat dan deze wijze van voorstellen een besparing oplevert als $3k < mn$, waarbij k het aantal onderling verschillende elementen is. Wanneer is sprake van een besparing als de waarden reëel zijn? [Bedenk dat een reëel getal twee woorden geheugenopslagruimte gebruikt.]
Als M_G de nabijheidsmatrix is van een digraaf met n knopen en m kanten, hoe kan deze dan efficiënt worden opgeslagen als m aanmerkelijk kleiner is dan n?

22. Een knoop v in een graaf $G = (V,E)$ met graad 1 heet een *blaadje*. Als $\#(V) > 1$ en G is een boom dan bevat V minstens twee blaadjes. Bewijs dit.

23. Een pad in een graaf $G = (V,E)$ waarin elke kant uit E precies één keer voorkomt heet een *Eulerpad*. Bewijs dat G een Eulercircuit bevat desd als
(a) er geen knooppunten met oneven graad voorkomen, óf
(b) als precies twee knopen oneven graad hebben.

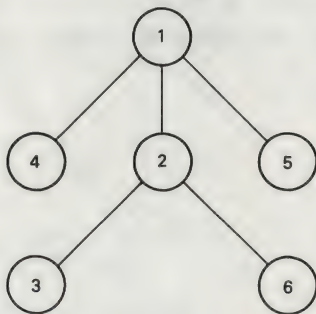
24. Als gegeven is een boom $T = (V,E)$ met $V = \{1,2,...,n\}$ dan wordt met behulp van de volgende procedure een string $w_T \in V^{n-1}$ uit T geconstrueerd.

begin $i := 1$;
  zolang $i \leqslant n-2$ *doe*
  begin
    bepaal $j \in V$ waarbij j het kleinste gehele getal is waarvoor geldt j is een blaadje;
    verwijder j en de bijbehorende tak e uit T;
    het i-de element uit de string w_T krijgt nu de waarde van het andere eindpunt van e;
    $i := i+1$
  eindezolang
einde

Bepaal de string uit $\{1,2,3,4,5,6\}$ die door middel van deze procedure wordt gegenereerd uit de boom in figuur 1.13. Beschrijf vervolgens een procedure die uit een string $w \in \{1,2,...,n^{n-2}\}$ een boom T voortbrengt met $w = w_T$. Pas deze procedure toe op de string 1123.

Toon aan dat er een een-eenduidige relatie bestaat tussen bomen met knopen $V = \{1,2,...,n\}$ en strings in V^{n-2}. Bewijs nu de volgende stelling van Cayley: het aantal verschillende opspannende bomen dat kan worden geconstrueerd over n verschillende knooppunten is n^{n-2}.

25. Toon aan dat de functie $g:\Gamma \to N$ die in dit hoofdstuk als een Gödelnummering werd gedefinieerd, inderdaad één op één (injectief) is. Bereken $g(G)$ voor de graaf in figuur 1.13. Welke eigenschap moet een Gödelgetal hebben opdat de graaf een boom is?



Figuur 1.13

2

Turingmachines

Tempt me no more; for I	Verzoek mij niet meer; want ik
Have known the lightning's hour,	Kende het uur van het weerlicht,
The poet's inward pride,	De innerlijke trots van de dichter,
The certainty of power.	De zekerheid door macht verschaft.

Cecil Day Lewis
Tempt me no more

DE FORMELE DEFINITIE

Teruggrijpend naar onze informele beschrijving in de inleiding kunnen we nu een *Turingmachine* (TM) definiëren als een 6-tupel
$M = (Q, \Sigma, T, P, q_0, F)$ met

(1) Q is een eindige verzameling *toestanden*,

(2) Σ is een eindige verzameling *symbolen*. Een symbool wordt weergegeven door $\wedge$ en heet het *lege symbool*.

(3) $T \subseteq \Sigma \setminus \{\wedge\}$ is de verzameling van *invoersymbolen*,

(4) P is een *programma* en dit is een partiële functie

$$(Q \setminus F) \times \Sigma \rightarrow Q \times \Sigma \times \{L, R, 0\} ,$$

(5) $q_0 \in Q$ is de *begintoestand*

(6) $F \subseteq Q$ is de verzameling van *eindtoestanden*.

Symbolen uit Σ kunnen op een tape voorkomen, dat wil zeggen ook symbolen die niet het lege symbool zijn en die niet in het invoeralfabet voorkomen. Deze symbolen heten *hulpsymbolen* en het bijbehorende *hulpalfabet* is gedefinieerd als $\Sigma \setminus T \setminus \{\wedge\}$.

Omdat P een partiële functie is, is $P(q, a)$, met $q \in Q \setminus F$ en $a \in \Sigma$ ofwel niet gedefinieerd, ofwel een uniek element uit $Q \times \Sigma \times \{L, R, 0\}$. Verder geldt dat $P(q, a) = (q', a', X)$ voor een $q' \in Q$ en een $a' \in \Sigma$

en $X \in \{L,R,0\}$ desd als het 5-tupel (q,a,q',a',X) in de programmatekst (zie de inleiding) voorkomt. P wordt gedefinieerd als een partiële functie om ervoor te zorgen dat het programma deterministisch is. Dat wil zeggen voor elke $q \in Q$ en $a \in \Sigma$ is er hoogstens één actie mogelijk van de TM als deze zich in toestand q bevindt en a als invoersymbool ontvangt.

In de inleiding gaven we al aan dat we Turingmachines meestal zullen weergeven door middel van gelabelde digrafen. Als $M = (Q,\Sigma,T,P,q_0,F)$ een TM is dan stellen we M voor door een gelabelde digraaf $G_M = (Q,E)$ met $E = \{(q_i,q_j)\}$ voor $a,a' \in \Sigma$ en $X \in \{L,R,0\}$ is $P(q_i,a) = (q_j,a',X)\}$. De verbinding tussen een knoop q_i en een knoop q_j wordt gelabeld met alle tripels (a,a',X) zodanig dat $P(q_i,a) = (q_j,a',X)$. Het labelen kan formeel worden beschreven door middel van een functie $l: E \to 2^{\Sigma\times\Sigma\times\{L,R,0\}}$ waarbij

$$l((q_i,q_j)) = \{(a,a',X) \mid P(q_i,a) = (q_j,a',X)\} \ .$$

Bij de weergave in diagramvorm van deze gelabelde digrafen zullen we de labels echter niet als verzamelingen weergeven. We laten de accolades en de komma's weg en vermelden alleen de tripelcomponenten, (zie figuur 2.4 en figuur 2.6b).

Hoe kunnen we nu een berekening door een TM formeel beschrijven? Veronderstel dat op een bepaald moment de lees/schrijfkop zich in toestand q bevindt en dat het symbool a op plaats i op de tape wordt gelezen. Zij verder λ de verste linker positie op de tape met een niet-leeg symbool en zij ρ de overeenkomstige verste rechter positie. Omdat de kop boven positie i staat, kunnen we de string links van de kop α definiëren door

$$\alpha = \begin{cases} \varepsilon & \text{als } i \leqslant \lambda \\ \text{de string gevormd door de symbolen op de plaatsen } \lambda \text{ tot } i-1, & \text{anders.} \end{cases}$$

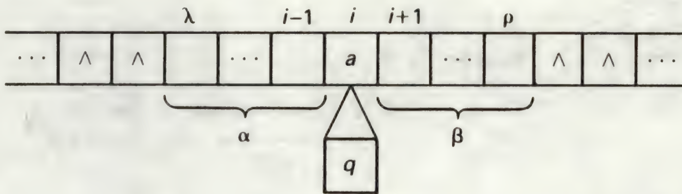
Evenzo kan β de string rechts van de lees/schrijfkop worden gedefinieerd als

$$\beta = \begin{cases} \varepsilon & \text{als } i \geqslant \rho \\ \text{de string gevormd door de symbolen op de plaatsen } i+1 \text{ tot } \rho, & \text{anders.} \end{cases}$$

De *configuratie* van een TM wordt nu gedefinieerd als het 5-tupel (q, i, α, a, β), zoals wordt geillustreerd in figuur 2.1.

De *initiële configuratie* C_0 hangt af van de input $x \in T^*$ en de begintoestand $q_0 \in Q$. Als $x = \varepsilon$ dan is $C_0 = (q_0, 1, \varepsilon, \wedge, \varepsilon)$. In alle andere gevallen is $x = ay$ met $a \in T$, $y \in T^*$ en $C_0 = (q_0, 1, \varepsilon, a, y)$. Samengevat is de configuratie dus

$$C_0 = (q\ , 1, \varepsilon, \textit{ als } x = \varepsilon \textit{ dan } \wedge \textit{ anders } kop(x), staart(x))$$



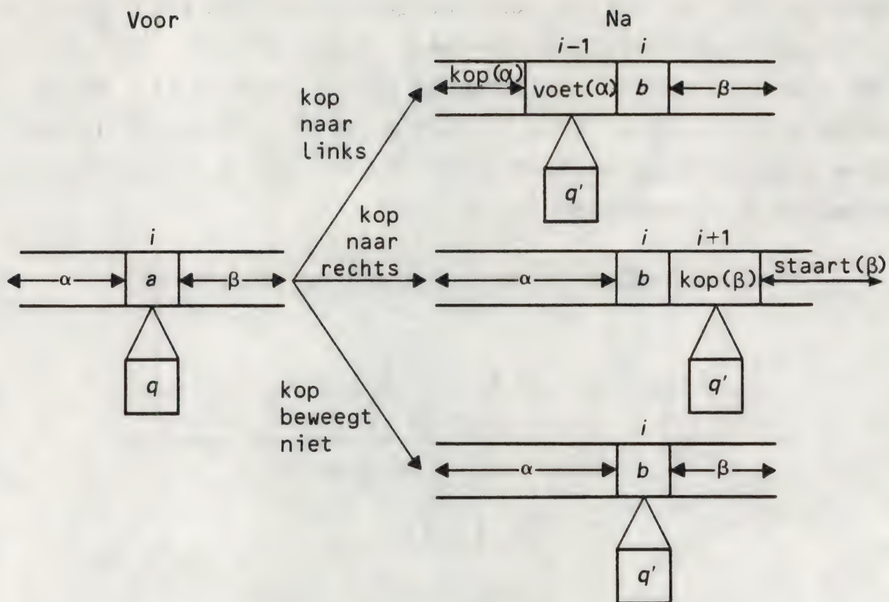
Figuur 2.1

Na de beginconfiguratie wijzigt de configuratie van een TM, *M* zich zoals door het programma wordt aangegeven. We geven deze rij van configuraties weer door $C_0, C_1, C_2, \ldots$ en noemen deze rij de *berekeningsrij* voor *M* bij input *x*.

Als $M = (Q, \Sigma, T, P, q_0, F)$ zich in configuratie $C_n = (q, i, \alpha, a, \beta)$ bevindt dan is de volgende configuratie C_{n+1} alleen gedefinieerd als $q \notin F$ en als $P(q, a)$ gedefinieerd is. In dat geval is

$$C_{n+1} = \begin{cases} (q', i-1, top(\alpha), \textit{ als } \alpha = \varepsilon \textit{ dan } \wedge \textit{ anders } voet(\alpha), b\beta) & \\ \qquad\qquad\qquad\qquad \text{als } P(q, a) = (q', b, R), & \\ (q', i+1, \alpha b, \textit{ als } \beta = \varepsilon \textit{ dan } \wedge \textit{ anders } kop(\beta), staart(\beta)) & \\ \qquad\qquad\qquad\qquad \text{als } P(q, a) = (q', b, R), & \\ (q', i, \alpha, b, \beta) \qquad \text{als } P(q, a) = (q', b, 0). & \end{cases}$$

In figuur 2.2 zijn de bijbehorende kopbewegingen aangegeven. Als de configuratie niet gedefinieerd is omdat $q \in F$ dan zeggen we dat de TM, *M stopt met succes*. In gevallen waarin geen volgende beweging is terwijl $q \notin F$ zeggen we dat de TM *stopt zonder succes*. In dit laatste geval is de output niet gedefinieerd, maar in het eerste geval definiëren we de output als de string $(\Sigma \setminus \{\wedge\})^*$ waarvan de symbolen



Figuur 2.2

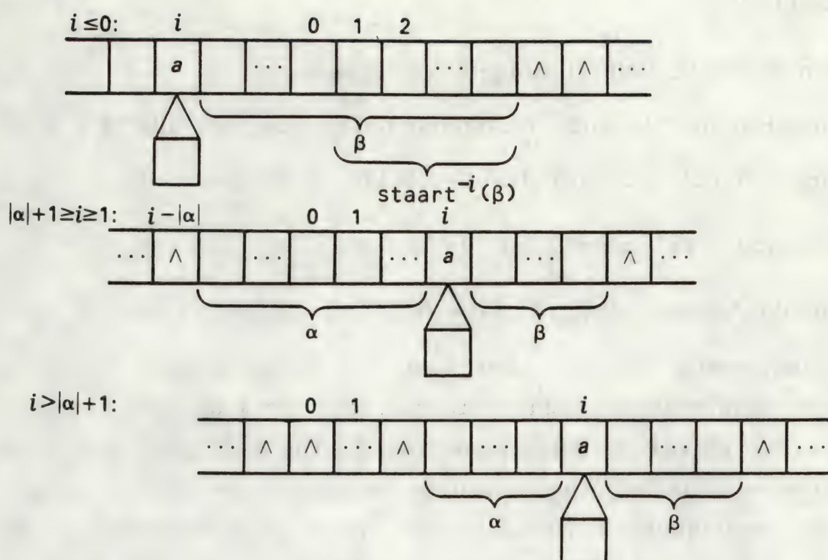
worden gevonden op de plaatsen 1 tot en met k-1 op de tape, waarbij $k \geqslant 1$ de verste linker positie is met het lege symbool. Veronderstel dat C_m de configuratie is waarin M met succes stopt. Om $output(C_m)$ formeel te kunnen definiëren moeten we eerst de functie $front: \Sigma^* \to (\Sigma \setminus \{\wedge\})^*$ als volgt definiëren

$$front(x) = \begin{cases} \varepsilon & \text{als } kop(x) = \wedge \text{ of } x = \varepsilon \\ concat(kop(x), front(staart(x))) & \text{anders} \end{cases}$$

De functie *front* accepteert een string $x = a_1 a_2 \ldots a_n$ uit Σ^* als argument en levert als waarde ofwel ε (als $x = \varepsilon$ of $a_1 = \wedge$), ofwel de string $a_1 a_2 \ldots a_{k-1}$ als a_k het eerste lege symbool in x is. Dus $front((011 \wedge 0 \wedge 100) = 011$ en $front(0001) = 0001$.

De functie $output(C_m)$ kunnen we nu formeel definiëren. Als $C_m = (q, i, \alpha, a, \beta)$ dan is

$$output(C_m) = \begin{cases} front(staart^{-i}(\beta)) & \text{als } i \geqslant 0 \\ front(staart^{|\alpha| - i + 1}(\alpha) a \beta) & \text{als } |\alpha| + 1 \geqslant i \geqslant 1 \\ \varepsilon & \text{anders} \end{cases}$$



Figuur 2.3

Deze drie mogelijke situaties worden geïllustreerd in figuur 2.3.

Veronderstel nu dat de TM, $M = (Q, \Sigma, T, P, q_0, F)$ een input $x \in T^*$ krijgt en dat hieruit een eindige berekeningsrij $C_0, C_1, C_2, \ldots, C_m$ volgt. Als M stopt met succes in de configuratie C_m dan is dit een *succesvolle berekeningsrij met lengte m* en we zeggen dat de TM, M *output*(C_m) bepaalt. De TM, M heeft dus met input x een resultaat berekend en dit resultaat noteren we als $f_M(x)$. Omdat M deterministisch is moet uit $x = y$ volgen dat $f_M(x) = f_M(y)$. Het hoeft echter niet zo te zijn dat f_M voor alle $x \in T^*$ gedefinieerd is. Dit kan komen omdat de machine stopt zonder succes, of omdat de machine helemaal niet stopt. De functie f_M is dus een partiële functie van T^* naar Σ^*. We noemen f_M de functie berekend door M. We noemen een functie $f: T^* \to \Sigma^*$ (*Turing*) *berekenbaar* desd als er een TM, M is zodanig dat $f = f_M$.

VOORBEELDEN

De volgende functies zijn Turing-berekenbaar:

(a) de partiële functie *ondef* die ongedefinieerd is voor alle $x \in \{0,1\}^*$

(b) de totale functie $cons0 : \{0,1\}^* \rightarrow \{0,1\}^*$ gedefinieerd door

$$cons0(x) = 0x \text{ voor alle } x \in \{0,1\}$$

(c) de totale functies $kop : \{0,1\}^* \rightarrow \{0,1\}$ en $staart : \{0,1\}^* \rightarrow \{0,1\}^*$.

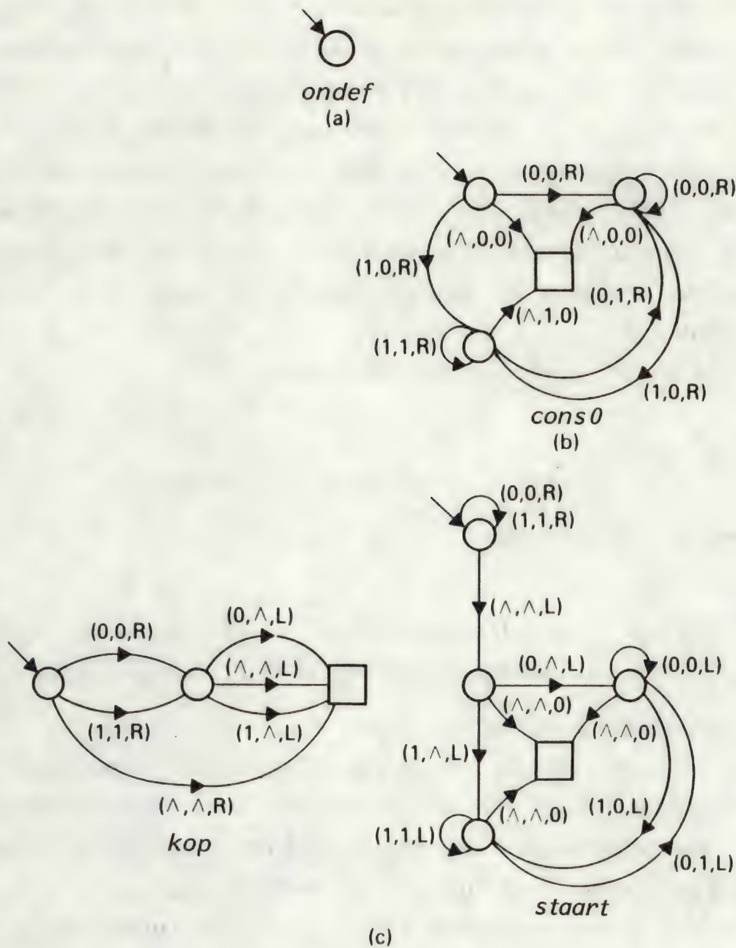
De bijbehorende TM's zijn weergegeven in figuur 2.4(a), (b) en (c). Net als in de inleiding hebben we ze getekend als gelabelde digrafen. Ga zorgvuldig na dat de tekeningen inderdaad correct zijn! We merkten al eerder op dat de benamingen van de toestanden er niet toe doen, zolang we verschillende toestanden maar kunnen onderscheiden. In de digrammen hebben we daarom meestal de toestanden naamloos gelaten. De begintoestand heeft een extra inkomende pijl en de eindtoestand geven we weer door een vierkantje in plaats van een cirkeltje.

Twee TM's $M_1 = (Q_1, \Sigma_1, T, P_1, q_{01}, F_1)$ en $M_2 = (Q_2, \Sigma_2, T, P_2, q_{02}, F_2)$ met hetzelfde input alfabet T heten *equivalent* desd als $f_{M_1}(x) = f_{M_2}(x)$ voor alle $x \in T^*$.

Een TM, $M = (Q, \Sigma, T, P, q_0, F)$ kan alleen zonder succes stoppen als er een $q \in Q$ en een $a \in \Sigma$ bestaat zo dat $P(q,a)$ niet gedefinieerd is. Deze situatie kan altijd worden vermeden door het construeren van een equivalente TM, $M' = (Q \cup \{q_d\}, \Sigma, T, P', q_0, F)$ waarbij $q_d \notin Q$ een 'fuik' of doodlopende toestand is. Als $q \in Q \backslash F$ en $a \in T$ dan is $P'(q,a) = P(q,a)$ als $P(q,a)$ gedefinieerd is en anders is $P'(q,a) = (q_d, a, 0)$. Als M' zich eenmaal in een fuik bevindt dan blijft M' in deze toestand omdat immers $P'(q_d, a) = (q_d, a, 0)$ voor alle $a \in \Sigma$. Duidelijk is dat M en M' equivalent zijn, maar waar M stopt zonder succes zal M' in de fuik terecht komen en daar voor altijd blijven. Hiermee hebben we de volgende stelling bewezen.

Stelling 2.1

Als f een partiële Turing-berekenbare functie is dan bestaat er een TM, M, zodanig dat $f_M = f$ en zo dat M alleen stopt in een eindtoestand, (dat wil zeggen M stopt altijd met succes).



Figuur 2.4

ANDERE TURING-BEREKENBARE FUNCTIES

We behandelen nu kort enkele berekenbare functies, gedefinieerd over de natuurlijke getallen N. Allereerst moeten we beslissen hoe we dergelijke gehele getallen binnen onze programma's zullen weergeven. Om de TM's eenvoudig te houden willen we een al te groot aantal symbolen in het inputalfabet zoveel mogelijk vermijden.

Zo zouden we $x \in N$ kunnen representeren in de *unaire notatie*,

dus als een string van enen. Het getal één wordt voorgesteld door 1, 2 door 11, 3 door 111 enzovoort. Een andere, aan alle informatici bekende, notatie is de *binaire notatie*. Het getal 1 wordt daarin voorgesteld als 1, 2 als 10, 3 als 11, 4 als 100 enzovoort. De unaire notatie is de allereenvoudigste, maar de binaire is meer algemeen gebruikelijk en ook korter. In de unaire notatie wordt $x \in N$ voorgesteld door een string met lengte x, maar in de binaire notatie is de lengte slechts $\lfloor \log_2 x \rfloor + 1$. Beide wijzen van weergave kunnen met behulp van recursieve functies worden gedefinieerd.

unairerep $N \rightarrow \{1\}^*$ wordt gedefinieerd door

$$unairerep(x) = \begin{cases} 1 & \text{als } x = 1 \\ concat(1, unairerep(x-1)) & \text{anders} \end{cases}$$

en *binairerep* $N \rightarrow \{0,1\}^*$ door

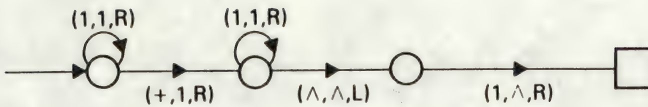
$$binairerep(x) = \begin{cases} 1 & \text{als } x = 1 \\ concat(binairerep(x\,'/'\,2), 0) & \text{als } x > 1 \text{ en even} \\ concat(binairerep(x\,'/'\,2), 1) & \text{als } x > 1 \text{ en oneven} \end{cases}$$

waarbij '/' geheeltallige deling voorstelt.

We zullen laten zien dat de keuze van de wijze van weergave geen invloed heeft op de inhoud van de klasse van Turing-berekenbare functies over N. Voorlopig kiezen we voor de unaire notatie en we bekijken de rekenkundige operaties optelling en vermenigvuldiging.

Stel we willen $x + y$ berekenen met $x, y \in N$. We coderen onze input als een string van enen ter lengte x gevolgd door een + en gevolgd door een string enen ter lengte y. We willen nu een TM construeren die, gegeven deze input, als resultaat oplevert de unaire representatie van $x + y$. Erg moeilijk is dat niet: we hoeven alleen maar de + string te verwijderen en de y enen die y weergeven één plaats naar links op te schuiven. Ook zouden we in plaats daarvan het plus-teken kunnen vervangen door een één en de meest rechtse één uit de string verwijderen. Deze laatste oplossing leidt tot de in figuur 2.5 getekende unaire opteller.

Vermenigvuldiging van twee getallen $x, y \in N$ is wat ingewikkelder. We veronderstellen dat de inputstring bestaat uit x enen, gevolgd door een maal-teken, gevolgd door y enen. Onze TM moet als resultaat van zijn berekening de unaire weergave van het produkt van x en y ople-



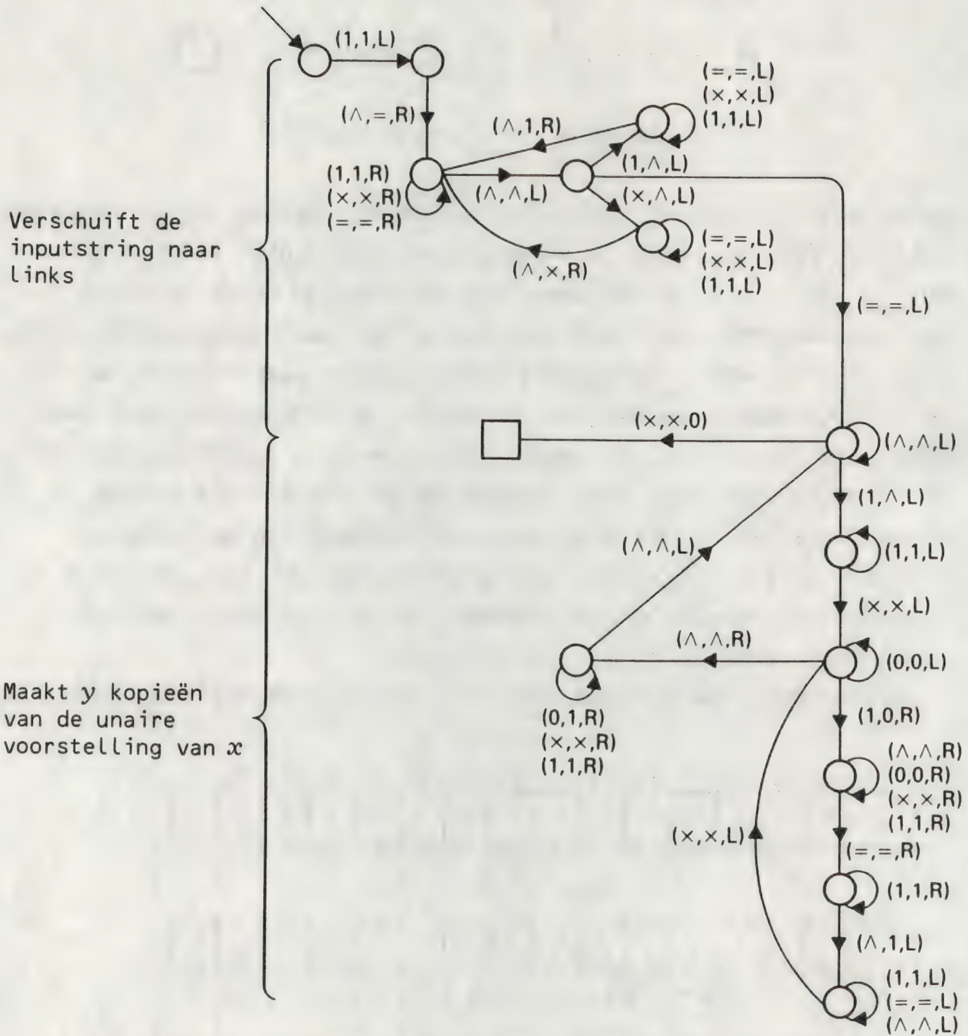
figuur 2.5 Een unaire opteller

veren. Om te beginnen verschuiven we de inputstring, die de plaatsen $1,2,...,k$ ($k = x+y+1$) op de tape inneemt, naar links, zodat de plaatsen $-k,-k+1,...,1$ de elementen van de string bevatten. Op deze manier kunnen we het positieve deel van de tape als werkruimte gebruiken zonder dat de oorspronkelijke inputstring wordt overschreven. We gaan nu de outputstring construeren. Initieel is de outputstring leeg, maar telkens als we een 1 uit de unaire weergave van y lezen, kopiëren we de unaire weergave van x en voegen die toe aan de outputstring. In figuur 2.6(a) hebben we deze werkwijze weergegeven en in figuur 2.6(b) is de TM getekend als een gelabelde digraaf. De symbolen 0 en = hebben we gebruikt als hulpsymbolen in het programma. Met het = symbool geven we plaats 0 op de tape aan.

Het is belangrijk het mechanisme van deze unaire vermenigvuldiger



Figuur 2.6(a) De berekening van 2×3



Figuur 2.6(b) Een unaire vermenigvuldiger

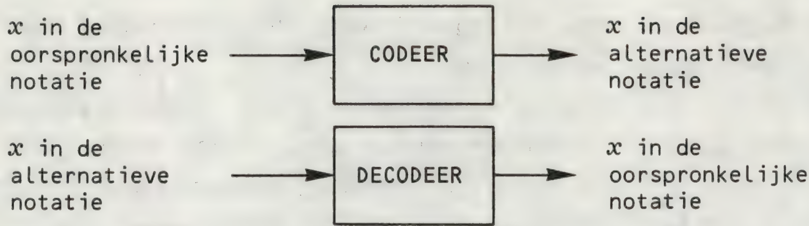
goed te begrijpen omdat het een aantal algemeen toepasbare technieken bevat, zoals het merken van een bepaalde plaats op de tape en het gebruik van hulpsymbolen tijdens het kopiëren. Later in dit hoofdstuk volgen nog meer voorbeelden van TM's. De bij de uitwerking gebruikte technieken zullen wij nog vaak toepassen. Ook bij het construeren van de TM's uit de oefeningen 2.1-2.4 moeten ze worden gebruikt. In de appendix staat een in Pascal geschreven programma vermeld dat een Turingmachine simuleert. Met behulp van deze simulator kan worden

gecontroleerd of een voor een bepaalde berekening ontworpen TM inderdaad de gewenste output oplevert.

TM's kunnen dus optellen en vermenigvuldigen. Herhaald gebruik van deze bewerkingen maakt het mogelijk aan te tonen dat elk polynoom $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ in N Turing-berekenbaar is. In oefening 2.3 wordt gevraagd te bewijzen dat ook aftrekking en geheeltallige deling berekenbaar zijn.

Tot nu toe gebruikten we de unaire notatie. Hoe zit het als we op een andere notatie voor onze getallen overgaan? Volgens de these van Church kunnen de berekeningen worden uitgevoerd onafhankelijk van de gebruikte notatie, zolang deze maar formeel gedefinieerd is. Dat betekent dat een getal x in een bepaalde notatie met behulp van een TM kan worden omgerekend naar een andere notatie en eveneens dat er een TM moet bestaan die de omgekeerde bewerking weer kan uitvoeren. In figuur 2.7 is dit schematisch weergegeven. Zij nu f de een of andere over de gehele getallen gedefinieerde functie en zij T_f een TM die f berekent met input en output in de oorspronkelijke notatie. We kunnen nu een TM T'_f construeren die eveneens f berekent, maar met input en output in een een andere notatie. Het enige dat T'_f behoeft te doen is eerst DECODEER los te laten op de inputstring, vervolgens T_f op te starten en tenslotte CODEER toe te passen op de outputstring.

Ter illustratie zullen we CODEER- en DECODEER-machines bouwen voor de unaire/binaire weergave. Allereerst de DECODEER-machine. Deze machine accepteert als input de binaire weergave van een natuurlijk getal en berekent de overeenkomstige unaire weergave. Omdat we de natuurlijke getallen beschouwen is de input minstens één en de output dus ook. Na deze eerste 1 lezen we van links naar rechts verder en daarbij gaan we als volgt te werk. Als we een 0 lezen in de inputstring dan verdubbelen we de outputstring en als we een 1 lezen dan doen we dat ook, maar we voegen er nog één extra 1 aan toe. In figuur 2.8 wordt deze werkwijze geillustreerd, waarbij we ervan uitgingen dat de inputstring al uit de plaatsen 1,2,..., werd gekopieerd naar de plaatsen - ,...,-2,-1 en dat plaats 0 werd gemerkt met het hulpsymbool =. In figuur 2.9 is de volledige machine als een gelabelde digraaf weergegeven.



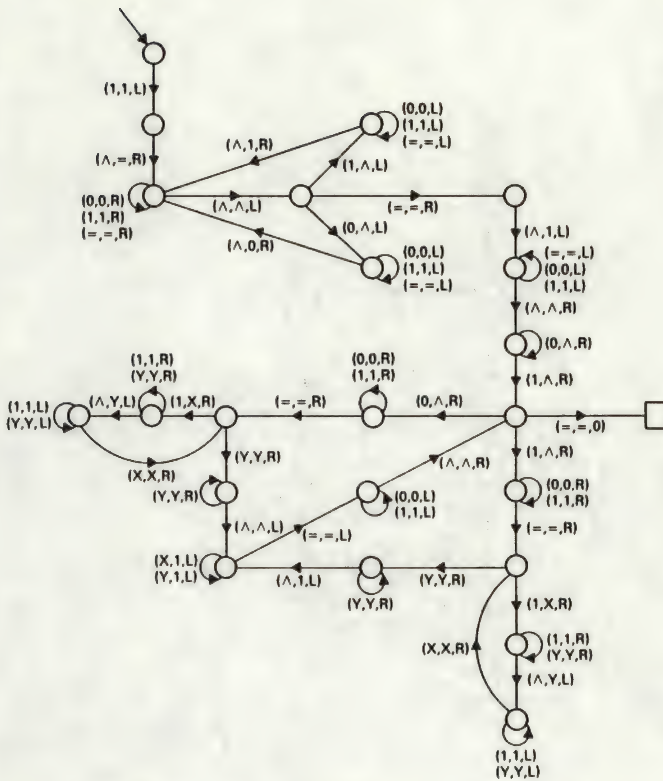
Figuur 2.7

···	−4	−3	−2	−1	0	1	2	3	4	5	6	···
···	∧	1	0	1	=	∧	∧	∧	∧	∧	∧	···
···	∧	∧	0	1	=	1	∧	∧	∧	∧	∧	···
···	∧	∧	∧	1	=	1	1	∧	∧	∧	∧	···
···	∧	∧	∧	∧	=	1	1	1	1	1	∧	···

Figuur 2.8 Conversie van de weergave van 5

Laten we nu eens proberen een machine te ontwerpen voor conversie van unair naar binair. De input is een string van $k > 0$ enen. Als $k = 1$ dan is de outputstring ook een 1. Als $k > 1$ dan moeten we nagaan of k even of oneven is. Als k even is dan is de outputstring gelijk aan de binaire voorstelling van k'/'2 gevolgd door een 0. Is k oneven dan is de outputstring gelijk aan de binaire voorstelling van k'/'2 gevolgd door een 1. ('/' is het teken voor geheeltallige deling.)

Om te beginnen vervangen we de eerste 1 en vervolgens elke tweede 1 in de inputstring door het hulpsymbool X. Als dit proces eindigt met het vervangen van een 1 door een X dan bevatte de inputstring een oneven aantal enen. Zo niet dan was het aantal enen even. In het eerste geval plaatsen we als meest rechtse symbool in de outputstring een 1, in het tweede geval een 0. We hebben nu nog precies k'/'2 enen in de inputstring over! We kunnen het proces nu opnieuw toepassen en een volgend symbool aan de outputstring toevoegen. Als we de laatste 1 in de inputstring door een X hebben vervangen dan zijn we klaar en in de binaire weergave compleet.

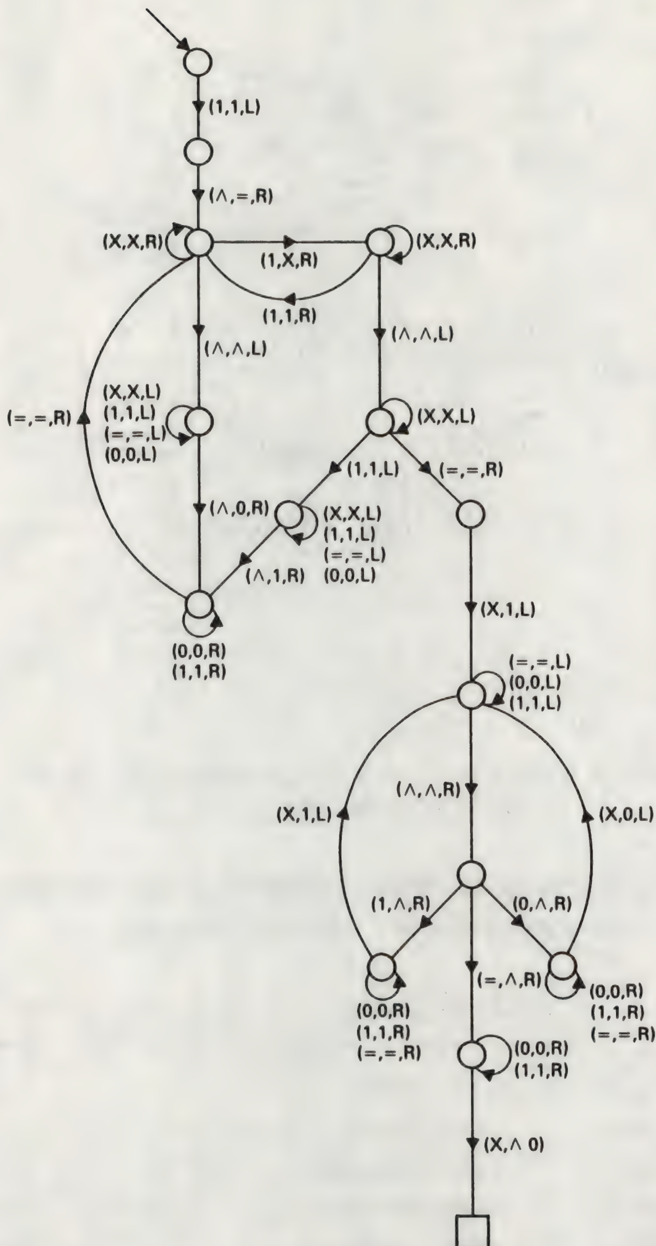


Figuur 2.9 Een machine voor conversie van binaire naar unaire notatie

In onderstaand voorbeeld is de inputstring de unaire weergave van het getal elf en is het conversieproces volledig uitgewerkt.

	input string	output string
input	11111111111	ε ε
na eerste iteratie	X1X1X1X1X1X	1
na tweede iteratie	XXX1XXX1XXX	11
na derde iteratie	XXXXXXX1XXX	011
na laatste iteratie	XXXXXXXXXXX	1011

In figuur 2.10 is de TM getekend die de conversie van unair naar binair uitvoert. De oorspronkelijke inputstring staat op plaatsen $1,\ldots,k$ op de tape. Plaats 0 is gemerkt met het hulpsymbool = en de



Figuur 2.10 Een machine voor conversie van unaire naar binaire weergave

output wordt symbool voor symbool geschreven naar de plaatsen -1,-2,.... Altijd is er na de voorlaatste iteratie precies één 1 in de inputstring over. Deze wordt bij de laatste iteratie vervangen door een X. Vervolgens schrijven we een 1 op plaats 1 en kopiëren, beginnend bij het meest linkse symbool, de elementen uit de negatieve tapeposities naar de plaatsen 2,3,....

EEN NIET-BEREKENBARE FUNCTIE

Zij $f:N \to N$ een Turing-berekenbare partiële functie, dan mogen we aannemen dat er een Turingmachine $T_f = (Q, \Sigma, \{1\}, P, q_0, F)$ bestaat die f in unaire notatie berekent. Ook mogen we altijd veronderstellen dat de toestanden van Q $q_1, q_2, \ldots$ zijn en dat de symbolen in Σ $1 = s_0, \wedge = s_1, s_2, s_3, \ldots$ zijn. Q is dan een eindige deelverzameling van de aftelbare verzameling $\bar{Q} = \{q_i | i \geqslant 0\}$ en Σ is een eindige deelverzameling van de aftelbare verzameling $\bar{\Sigma} = \{s_i | i \geqslant 0\}$. We kunnen T_f eenvoudig beschrijven door het weergeven van zijn programma als een lijst van 5-tupels en door aan te geven welke van de toestanden eindtoestanden zijn. Elk 5-tupel in het programma is van de vorm $t = (q_i, s_j, q_k, s_l, X)$. De indexen i, j, k en l zijn niet-negatieve gehele getallen en $X \in \{L, R, 0\}$. Een 5-tupel t kan worden gecodeerd als een geheel getal $g(t)$ met behulp van de Gödel aftellingstechniek,

$$g(t) = 2^{i+1} 3^{j+1} 5^{k+1} 7^{l+1} 11^{m+1}$$

waarbij $m = 0$ als $X = L$, $m = 1$ als $X = R$ en $m = 2$ als $X = 0$. Een eindige rij van 5-tupels, $\underset{\sim}{t} = (t_1, t_2, \ldots, t_n)$ kan nu worden gecodeerd als het gehele getal

$$g(\underset{\sim}{t}) = 2^{g(t_1)} 3^{g(t_2)} \ldots p_n^{g(t_n)}$$

Hierbij is p_n het n-de priemgetal. Als nu de TM T_f een programma heeft dat wordt beschreven door de rij 5-tupels $(t_1, t_2, \ldots, t_n)$ en door de eindtoestanden $q_{\lambda 1}, q_{\lambda 2}, \ldots, q_{\lambda r}$ $\lambda 1 < \lambda 2 < \ldots < \lambda r$ dan kunnen we T_f coderen als

$$g(T_f) = 2^{g(L)} 3^{\lambda 1+1} 5^{\lambda 2+1} \ldots p_r^{\lambda r+1}$$

Met het vaststellen van deze Gödelaftelling hebben we de volgende stelling bewezen:

Stelling 2.2

De verzameling $\{ T_f | T_f$ is een TM van het hierboven beschreven type en berekent een partiële functie $f:N \rightarrow N\}$ is aftelbaar oneindig.

Als een verzameling aftelbaar is dan kunnen haar elementen systematisch worden opgesomd: $T_{f_1}, T_{f_2}, \ldots, T_{f_i}$ berekent $f_i:N \rightarrow N$. Elke berekenbare functie f_i wordt berekend door minstens één T_{f_i} uit deze lijst. We gebruiken nu dezelfde techniek als bij Cantor's diagonalisatiestelling (zie Hoofdstuk 1) om een functie $g:N \rightarrow N$ te construeren die niet door een T_g in de lijst wordt berekend. We definiëren g eenvoudig als volgt

$$g(n) = \begin{cases} f_n(n)+1 & \text{als } f_n(n) \text{ gedefinieerd is} \\ 1 & \text{als } f_n(n) \text{ niet gedefinieerd is} \end{cases}$$

Veronderstel nu dat bij g een Turingmachine uit de lijst hoort. Dat zou betekenen dat g door een of andere T_{f_k} berekend wordt. T_{f_k} berekent echter f_k en op grond van de definitie is in ieder geval $g(k) \neq f_k(k)$ en dus is $g \neq f_k$. Onze veronderstelling leidt dus tot een tegenspraak en dus is g niet berekenbaar. Hiermee hebben we bewezen:

Stelling 2.3

Er bestaan niet berekenbare functies $N \rightarrow N$.

De situatie is zelfs nog ernstiger dan de stelling suggereert! In oefening 16 in hoofdstuk 1 werd gevraagd te bewijzen dat er onaftelbaar veel partiële functies $N \rightarrow N$ bestaan. Hier toonden we aan dat daarvan maar aftelbaar veel Turing-berekenbaar zijn en dus moeten er onaftelbaar veel niet-berekenbare functies bestaan. Een tamelijk bedroevend resultaat, maar geen aanklacht tegen de Turingmachine. Tot nu

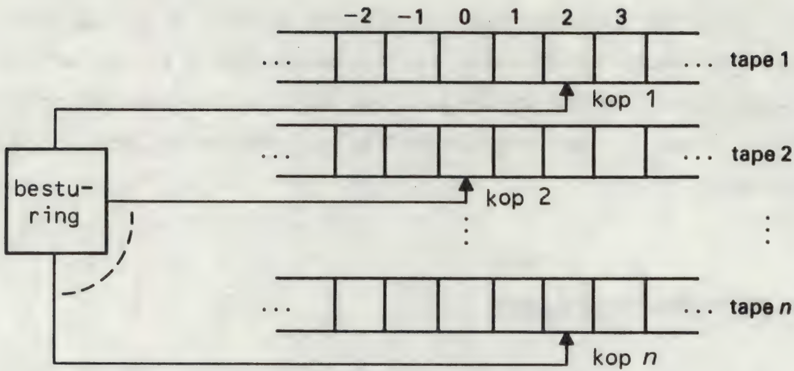
toe wijst alles erop dat de these van Church correct is en dat elke effectief berekenbare functie ook Turing-berekenbaar is. Nu moeten we echter constateren dat het merendeel van alle functies niet effectief berekenbaar is en dus wel een theoretische, maar zeker geen praktische betekenis heeft.

MULTITAPE TURINGMACHINES

Als we de these van Church voor waar houden, (en dat doen we!) dan kunnen Turingmachines elke functie berekenen die door enig andere machine berekend kan worden. Dat betekent ook dat het uitbreiden van een Turingmachine met toeters en bellen niet zal leiden tot vergroting van de rekencapaciteit. In dit hoofdstuk lichten we deze bewering toe met behulp van Turingmachines met meer dan één tape. In hoofdstuk 4 volgt een nadere toelichting aan de hand van niet-deterministische Turingmachines, (zie verder ook opgave 2.10).

Een multitape Turingmachine heeft $n>1$ eindeloze tapes en n lees/schrijfkoppen. De koppen worden centraal bestuurd zoals is weergegeven in figuur 2.11. Het besturingsmechanisme kan zich in een eindig aantal toestanden uit een verzameling Q bevinden. Het programma specificeert de volgende stap die steeds afhangt van de toestand q van het besturingsmechanisme en van de op de diverse tapes gelezen symbolen. De machine kan nu overgaan naar een nieuwe toestand, symbolen op één of meer tapes schrijven en één of meer van de koppen verplaatsen naar links of naar rechts.

We zullen tape 1 voor input en output gebruiken; de overige tapes vormen dus een uitbreiding van de werkruimte. Om te beginnen wordt een inputstring $x \in T^*$ geschreven naar de plaatsen $1,2,\ldots,|x|$ van tape 1. Alle andere plaatsen op deze en de andere tapes zijn leeg. Alle koppen beginnen met het lezen van de inhoud van plaats 1 van hun tape. De machine zal stoppen als het besturingsmechanisme in een eindtoestand uit $F \subseteq Q$ terecht komt. De output wordt naar tape 1 geschreven. Als plaats 1 van tape 1 het lege symbool bevat dan is de output de lege string ε, anders is de output gelijk aan de string gevormd door de inhouden van de plaatsen $1,2,\ldots,k-1$ van tape 1, waarbij k de meest linkse locatie is die het lege symbool bevat, $(k > 0)$. Ook hier



Figuur 2.11 Een multitape Turingmachine

houden we de mogelijkheid open dat er van hulpsymbolen gebruik wordt gemaakt en de output is dus algemeen een element van de verzameling Σ^* met $T \subseteq \Sigma$.

We zullen de multitape TM niet strikt formeel behandelen, maar de informele behandeling zal zodanig zijn dat de lezer desgewenst zelf het betoog kan formaliseren. We geven nu een schets van het bewijs van de volgende stelling.

Stelling 2.4

Een functie $f: T^* \to \Sigma^*$ is Turing-berekenbaar desd als de functie berekenbaar is op een multitape Turingmachine.

Schets van het bewijs

$\Rightarrow$: Het bewijs in deze richting is triviaal: als de functie Turing-berekenbaar is dan behoeft de multitape machine alleen maar alle berekeningen op zijn eerste tape uit te voeren, zoals gespecificeerd door het programma voor de één-tape machine.

$\Leftarrow$: We geven een schets van het omgekeerde bewijs voor het geval waarin f berekend wordt door een 2-tape TM, M. Zelfs de schets van het bewijs is tamelijk gecompliceerd en deze kan desnoods bij eerste lezing worden overgeslagen. Het vervolgens generaliseren van het

resultaat naar een machine met n tapes is daarentegen eenvoudig en wordt als oefening aan de lezer overgelaten.

We gaan uit M een één-tape TM, M' construeren die f ook berekent. Veronderstel dat Σ het tape-alfabet van M is. Het tape-alfabet van M' laten we dan bestaan uit tripels (X, B_1, B_2), waarbij X het symbool 0 of het symbool 1 is en waarbij B_1 en B_2 elementen uit Σ zijn, maar mogelijk met superscript 1. We gaan nu een bepaalde configuratie waarin M zich bevindt nabootsen door middel van een configuratie van M'. Laten we een willekeurige tapelocatie k beschouwen en veronderstel dat tape 1 van M hier het symbool A_1 heeft staan en tape 2 het symbool A_2. De inhoud van plaats k van M' wordt nu bepaald door het tripel (X, B_1, B_2), X heeft de waarde 1 als $k = 1$ en in alle andere gevallen is $X = 0$. Verder is $B_i = A_i^1$ $(i = 1,2)$ als kop i plaats k van tape i leest, terwijl $B_i = A_i$ als dat niet het geval is.

Als nu de één-tape machine M' als input de string $x = a_1 a_2 \ldots a_n$ $(n > 0)$ ontvangt dan zullen we M' allereerst deze input laten lezen en laten vervangen door tripels $(1, a_1^1, \wedge^1), (0, a_2, \wedge), \ldots, (0, a_n, \wedge)$. Als $n = 0$ dan vullen we plaats 1 met het tripel $(1, \wedge^1, \wedge^1)$. Het lege symbool van M' geven we aan met $(0, \wedge, \wedge)$ en we veronderstellen dat alle overige plaatsen op de tape dit lege symbool bevatten. Op deze wijze bevat de tape van M' de initiële inhouden van de tapes van M.

Nu moeten we het mechanisme van de TM M' beschrijven dat de acties van de multitape machine M nabootst. Als Q de toestandverzameling van M is dan is de toestandverzameling van M' weer te geven met $Q \times \Sigma^2 \times 2^{\{1,2\}} \times \{<, =, >\} \times \{L, 0, R\}^2$. M' is in de toestand $(q, (A_1, A_2), S, X, (Y_1, Y_2))$ desd als voor de configuratie van M geldt:

(i) M is in toestand q

(ii) kop_1 leest A_1 en kop_2 leest A_2.

Elke locatie op de tape van M' die één of meer componenten bevat met superscript 1 heet een *actieve locatie*. Maximaal zijn er twee actieve locaties; de eerste heeft een superscript boven zijn tweede component en de tweede boven zijn derde component. De eerste component van de toestand van M' wordt gebruikt voor het bepalen van de plaats van actieve locaties op de tape. Als de kop van M' plaats k op de tape leest dan geldt $i \in S$ voor $i = 1,2$ desd als de i-de locatie zich links van plaats k bevindt. We gebruiken het symbool X uit de toestandsbeschrij-

ving om aan te geven of $k < 1$, $k = 1$ of $k > 1$. De symbolen (Y_1, Y_2) worden gebruikt bij het simuleren door de machine M' van één stap van de machine M. In figuur 2.12 wordt de constructie die we hier beschreven nog eens door middel van een plaatje weergegeven.

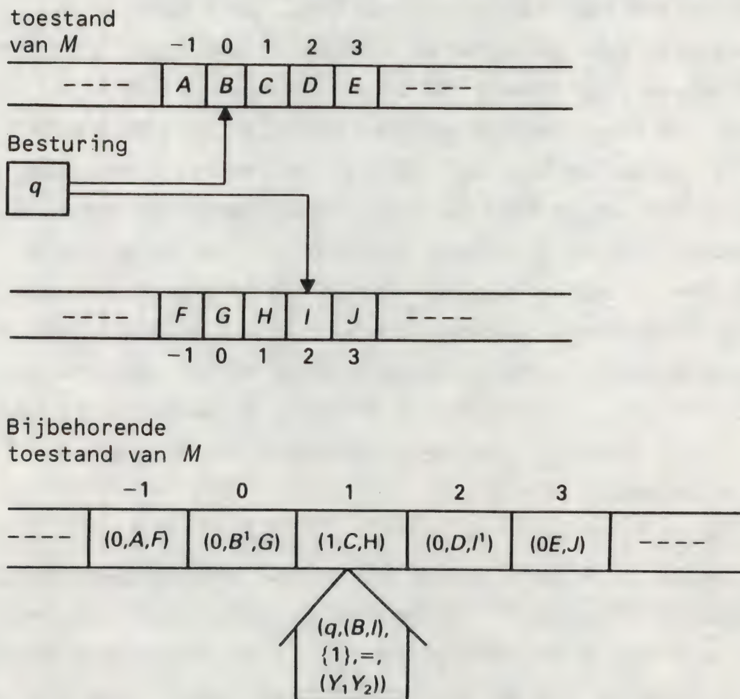
Nadat M' de inputstring heeft vervangen op de hierboven beschreven manier, beweegt M' zijn lees/schrijfkop terug naar de plaats 1 op de tape, (die terug te vinden is omdat zijn eerste component een 1 bevat), en komt in de toestand $(q_0,(a_1,\wedge),\phi,=,(0,0))$. Het simuleren van M kan nu beginnen.

De simulatie van een stap van M wordt door M' in twee fasen uitgevoerd. In de eerste fase berekent M' uit zijn huidige toestand:

(i) de volgende toestand van het besturingsmechanisme van M

(ii) de symbolen die door de koppen van M geschreven zullen worden

(iii) de richtingen waarin de koppen van M zich zullen bewegen.



Figuur 2.12

Deze gegevens worden door M' opgeslagen in de eerste, tweede en vijfde component van zijn toestandsbeschrijving. Nu begint de tweede fase

van de berekening. Hiervoor moet de lees/schrijfkop de twee actieve locaties achtereenvolgens bezoeken. Terwijl M' zijn tape doorloopt moeten de derde en vierde component van zijn toestandsbeschrijving worden aangepast. Op grond van de informatie in de derde component kan M' vaststellen in welke richting de volgende actieve locatie moet worden gezocht. Als M' de i-de ($i = 1,2$) actieve locatie vindt dan krijgt de (i+1)-ste component van het daar opgeslagen tripel het superscript 1. Dit symbool moet vervolgens vervangen worden door de i-de component van de tweede component van de toestandsbeschrijving van M'. Nu wordt het superscript 1 geschreven bij ofwel de (i+1)-ste component van het zojuist gelezen tripel, ofwel van het tripel links daarvan, ofwel van het tripel rechts daarvan, afhankelijk van de waarde 0, L of R van de i-de component van de vijfde component van de toestandsbeschrijving van M'. De tweede fase van de simulatie van een stap van M is pas voltooid als beide actieve locaties op de tape van M' zijn bezocht en als de bijbehorende componenten zijn aangepast. Tijdens dit bezoekproces kan ook de tweede component van de toestand van M' worden aangepast zodat deze vervolgens het nieuwe paar symbolen dat door de koppen van M gelezen wordt bevat.

Als fase twee is doorlopen dan is de simulatie van een stap van M door M' voltooid. Dit proces wordt nu zo vaak als nodig is herhaald teneinde M' de berekening van M te laten nabootsen. Als M in een eindtoestand komt dan zal M' gebruik maken van zijn vierde component om plaats 1 op de tape terug te vinden. Als plaats 1 door M' is gevonden dan wordt de output gegenereerd door de tape naar rechts uit te lezen en elke gelezen tripel te vervangen door de tweede component van dat tripel. Echter zodra een tripel wordt gelezen met als tweede component $\wedge$ of $\wedge^1$ dan wordt het lege symbool $(0,\wedge,\wedge)$ op de tape geschreven en komt ook M' in zijn eindtoestand. Op deze wijze is ook de output van M', gegeven input x, juist $f(x)$ en is de simulatie voltooid.

BEPERKTE TURINGMACHINES

Hoewel het TM-model eenvoudig en betrekkelijk gemakkelijk toe te passen is, is het toch krachtig genoeg om er elke effectief berekenbare

functie mee te kunnen berekenen. In deze paragraaf brengen we een aantal restricties aan in het model met behoud van het oorspronkelijke rekenvermogen. Een mogelijke beperking in het model is de veronderstelling dat de gebruikte tape niet in beide richtingen oneindig lang is maar slechts in één richting. In dat geval zijn de locaties op de tape niet meer genummerd met ... -2,-1,0,1,2,... maar met 1,2,3,... . Het TM programma wordt op de gewone manier uitgevoerd, maar als de kop naar links zou moeten bewegen na het lezen van locatie 1 dan stopt de machine zonder succes. We noemen een op deze manier beperkte TM een *Turingmachine met een eenzijdig oneindige tape*. De oorspronkelijke TM is dan een Turingmachine met tweezijdig oneindige tape.

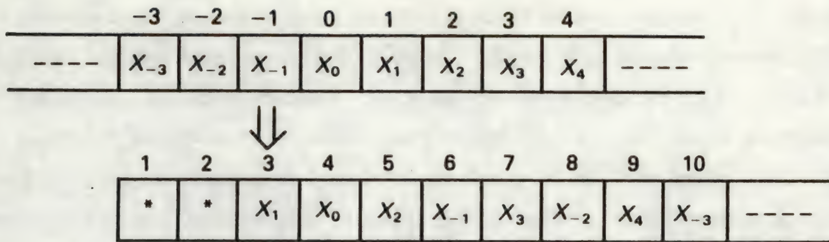
Stelling 2.5

Als $f: T^* \to \Sigma^*$ een TM-berekenbare partiële functie is dan is f ook berekenbaar op een Turingmachine met eenzijdig oneindige tape.

Bewijs. Ook hier geven we alleen een schets van het bewijs en laten we de formele bewijsvoering aan de lezer over. Omdat f TM-berekenbaar is, bestaat er een TM met tweezijdig oneindige tape $M = (Q, T, \Sigma, P, q_0, F)$ met $f = f_M$. We zullen laten zien hoe we uit M een Turingmachine met eenzijdig oneindige tape, M' kunnen construeren met $f_{M'} = f_M$.

De kern van het bewijs bestaat uit het weergeven van de inhoud van een tweezijdig oneindige tape op een eenzijdig oneindige tape. Hierbij gebruiken we dezelfde techniek als we in hoofdstuk 1 gebruikten om aan te tonen dat Z aftelbaar is. We beelden de inhoud van de tweezijdig oneindige tape af op de eenzijdig oneindige zoals aangegeven in figuur 2.13. Locaties 1 en 2 op de eenzijdig oneindige tape bevatten beiden het speciale symbool $* \notin \Sigma$. De bedoeling hiervan zullen wij nu verduidelijken. De inhoud van plaats $i \geqslant 1$ van M vinden we terug op plaats $2i + 1$ van M' en de inhoud van plaats $i \leqslant 0$ van M vinden we op plaats $4 - 2i$ van M'.

Veronderstel nu dat M plaats $i > 1$ leest; als M' op hetzelfde punt in de berekening is aangeland dan moet M' dus plaats $2i + 1$ lezen. Als M vervolgens het gelezen symbool door X vervangt en één plaats



Figuur 2.13

naar rechts opschuift, dan moet M' het gelezen symbool ook door X vervangen, maar twee plaatsen naar rechts opschuiven. Op dezelfde manier wordt een beweging naar links door M vervangen door een beweging twee plaatsen naar links door M'. Als P het programma van M is dan construeren we dus hieruit een programma P'_{R} voor M' door elke beweging naar links of naar rechts in P te vervangen door een beweging twee plaatsen naar links of rechts. We zullen veronderstellen dat de in P'_{R} gebruikte toestanden op een subscript R na gelijk zijn aan de toestanden in P. Zo kunnen we met P'_{R} het programma P nabootsen op het positieve deel van een tape.

Als M daarentegen een locatie $i < 0$ leest dan is de bijbehorende locatie van M' $4 - 2i$. In dat geval wordt een beweging naar links van M vertaald door een beweging twee plaatsen naar rechts van M' en een beweging naar rechts door M komt overeen met een beweging twee plaatsen naar links door M'. We construeren nu een programma P'_{L} uit P door een subscript L te plaatsen bij alle toestanden uit P en door elke beweging naar rechts te vervangen door een beweging twee plaatsen naar links en elke beweging naar links door een beweging twee plaatsen naar rechts.

Met de constructie van P'_{R} en P'_{L} hebben we het gewenste programma P' bijna geconstrueerd. Het programma P' zal allereerst zijn inputstring twee plaatsen naar rechts opschuiven en de eerste twee plaatsen op de tape met het symbool * vullen. Nu wordt de kop naar plaats 3 bewogen en wordt met het uitvoeren van P'_{R} begonnen. Als in een bepaald stadium van de berekening na een dubbele stap naar links het symbool * wordt gelezen dan weten we dat we ons op plaats 1 bevinden. Als M' zich nu in toestand q_{R} bevindt dan is de overeenkomstige toestand van M toestand q en leest M plaats 0. We definiëren nu $P'(q_{\mathrm{R}}, *)$

zo dat de kop van M' drie plaatsen naar rechts wordt bewogen en zich dus boven locatie 4 bevindt. Tegelijkertijd zorgen we dat hierbij toestand q_L van P'_L behoort. Nu gaat M' verder met het uitvoeren van P'_L totdat na een dubbele beweging naar links opnieuw een * wordt gelezen. Dit betekent dat we ons op locatie 2 bevinden en M' moet nu één plaats naar rechts bewegen, de inhoud van locatie 3 lezen en overgaan van een toestand $q_L \in P'_L$ naar de bijbehorende $q_R \in P'_R$.

Het symbool * gebruiken we dus om de overgang van P'_R naar P'_L en omgekeerd te markeren en zo simuleert P' het gedrag van P. Nu is er nog één aanpassing nodig om ervoor te zorgen dat P' de juiste output genereert. Als P in een eindtoestand komt dan moet P' de inhoud van zijn tape aanpassen door de inhoud van de plaatsen 3,5,7,... te kopiëren naar de plaatsen 1,2,3,... tot en met het eerste lege symbool. Hiermee is de constructie van P' gereed.

In de meeste gevallen zullen we ons houden aan het oorspronkelijke TM model met de tweezijdig oneindige tape. Het speciale geval waarin de machine stopt zonder succes na een poging naar links te bewegen vanuit plaats 0 hoeven we dan niet in aanmerking te nemen. Bepaalde bewijzen kunnen echter worden vereenvoudigd door uit te gaan van het model met de eenzijdig oneindige tape.

Er zijn andere restricties binnen het TM-model mogelijk die het rekenvermogen niet aantasten. Een goed overzicht van deze restricties staat in Fischer (1965). We kunnen bijvoorbeeld uitgaan van een TM met twee tapes, waarbij de eerste tape wordt gebruikt voor input en output, terwijl de tweede tape als werktape wordt gebruikt. We kunnen nu bijvoorbeeld eisen dat behalve het lege symbool slechts één ander symbool op de werktape wordt gebruikt. Dat deze machine equivalent is met een normale TM kan vrij eenvoudig worden aangetoond. Stel $M = (Q, \Sigma, T, P, q_0, F)$ is weer de oorspronkelijke TM met n mogelijke niet-lege symbolen. Het i-de symbool kunnen we nu op de werktape coderen als $1^i \wedge^{n-i}$ en het lege symbool door $\wedge^n$. Elk symbool uit wordt zo uniek weergegeven binnen precies Σ locaties op de tape. De 2-tape TM codeert nu eerst de input van zijn eerste tape naar zijn tweede tape met behulp van deze codering. Nu kan de werking van worden gesimuleerd op de tweede tape met de gecodeerde symbolen.

Als de simulatie stopt dan kan de output worden geconstrueerd door de informatie op de werktape te decoderen en terug te schrijven naar de output-tape.

Een soortgelijke eigenschap die we later nodig zullen hebben en die is gebaseerd op binaire codering is geformuleerd in de volgende stelling.

Stelling 2.6

Elke Turing-berekenbare functie $\{0,1\}^* \rightarrow \{0,1\}^*$ kan worden berekend met een één-tape TM die slechts $\{0,1,\wedge\}$ als tape-alfabet gebruikt.

Bewijs (schets). Stel dat $M = (Q,\Sigma,\{0,1\},P,q_0,F)$ de functie $f:\{0,1\}^* \rightarrow \{0,1\}^*$ berekent en dat Σ de symbolen $0,1,s_1,s_2,\ldots,s_n$ als als niet-lege symbolen bevat, waarbij $n \geqslant 1$. Elk van deze symbolen kan worden gecodeerd als een string in $\{0,1\}^*$ ter lengte $k = \lceil \log_2(n+2) \rceil$ met behulp van een binaire code. Als bijvoorbeeld $n = 5$ dan is $k = 3$ en coderen we 0 als 000, 1 als 001, s_1 als 010, s_2 als 011, s_3 als 100, s_4 als 010 en s_5 als 110. De code 111 wordt in dit geval niet gebruikt.

We construeren nu weer een TM, M' uit M. M' codeert eerst de input met codes uit $\{0,1\}^*$, zoals hierboven aangegeven. M' simuleert vervolgens de werking van M en als deze simulatie stopt dan kan M' de informatie op de tape decoderen naar de gewenste output. Tijdens dit coderen kan M' alle coderingen van symbolen s_i vertalen naar het lege symbool $\wedge$, omdat het resultaat immers alleen symbolen uit $\{0,1\}$ mag bevatten. Als elke codering van de symbolen 0 en 1 correct naar 0 en 1 wordt vertaald dan genereren we de correcte output en wordt ervoor gezorgd dat M' slechts de tekens $0,1$ en $\wedge$ op zijn tape heeft staan.

OEFENINGEN

1. Construeer een TM om *lengte*: $\{0,1\}^+ \rightarrow N$ te berekenen met de output in unaire notatie. Construeer hieruit, of op een andere manier, een TM die lengte berekent met de output in binaire notatie.

2. Construeer een TM ter berekening van $rev: \{0,1\}^* \to \{0,1\}^*$ zoals gedefinieerd in opgave 18 uit hoofdstuk 1.

3. Gegeven is de functie $palindroom: \{0,1\}^* \to \{0,1\}$ met

$$palindroom(x) = \begin{cases} 1 & \text{als } x = rev(x) \\ 0 & \text{anders} \end{cases}$$

Laat zien dat $palindroom(x) = 1$ desd als ofwel $lengte(x) \leqslant 1$, ofwel x is van de vorm $0y0$ of $1y1$ met $palindroom(y) = 1$. Construeer nu een TM voor het berekenen van *palindroom*. Is deze methode efficienter of minder efficiënt dan direct $rev(x)$ construeren en nagaan of $x = rev(x)$?

4. Bouw een TM voor het berekenen van *sub* en $div: N \times N \to N$, waarbij

$$sub(x,y) = \begin{cases} x - y & \text{als } x \geqslant y \\ \text{onbepaald} & \text{anders} \end{cases}$$

en

$$div(x,y) = \begin{cases} k & \text{als } x = ky \text{ voor een } k \in N \\ \text{onbepaald} & \text{anders} \end{cases}$$

Veronderstel in beide gevallen dat de input bestaat uit een unaire representatie van x gevolgd door een symbool voor de bewerking (- of :), gevolgd door een unaire representatie van y.

5. Pas de antwoorden op opgave 4 zo aan, dat de Turingmachines nooit zonder succes stoppen als de input uit $\{-,:,1\}^*$ is.

6. Controleer uw antwoorden op de opgaven 1, 2 en 3 door de Turingmachines te simuleren met de TM-simulator uit de Appendix. Kies steeds een input testset die het programma volledig uittest.

7. Codeer de TM-programma's uit opgave 4 als quintupels en voer ze in in de TM-simulator. Beschrijf de input-testset waarmee u de juistheid of (onjuistheid!) van uw programma's denkt te kunnen aantonen. (De TM-simulator kan in het vervolg steeds worden gebruikt om programma's te testen.)

8. Definieer een functie $\{0,1\}^* \to \{0,1\}^*$ die *niet* Turing-berekenbaar is.

9. Als we als input aan de unaire opteller *m* enen geven, gevolgd door een +, gevolgd door *n* enen, wat is dan de lengte van de output van een succesvolle berekening? En wat is het overeenkomstige resultaat voor de unaire vermenigvuldiger? Hoe kan de gemiddelde efficiëntie van deze programma's worden verbeterd?

10. Stelt u zich een TM voor met een twee dimensionale tape. Een plaats op de 'tape' wordt nu bepaald door de coördinaten (i,j) met $i,j \in Z$. Afhankelijk van zijn toestand en het juist gelezen symbool kan deze *twee-dimensionale Turingmachine* naar een nieuwe toestand overgaan en daarbij zijn lees/schrijfkop in één van vier mogelijke richtingen (links, rechts, op, neer) bewegen. Op ieder tijdstip staat er slechts een eindig aantal niet-lege symbolen op de tape. Inputs en outputs worden op analoge wijze als bij een één-dimensionale tape verwerkt op de plaatsen $(1,1),(1,2),(1,3),\ldots$. Toon aan dat elke functie die berekenbaar is met een twee-dimensionale TM, ook berekenbaar is met een gewone (één-dimensionale) TM.

3

Oplosbaarheid en Onoplosbaarheid

The troubles of our proud and angry dust
Are from eternity, and shall not fail.
Bear them we can, and if we can we must.
Shoulder the sky, my lad, and drink your ale.

A.E. Housman, Last Poems

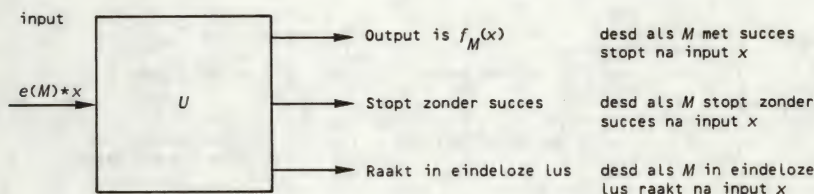
Moeite met ons trots onstuimig bloed
Is voor eeuwig en altijd ons deel.
Maar wij kunnen verdragen en wie kan
die moet.
Schouders onder de hemel, kerel en
giet je bier door je keel.

EEN UNIVERSELE TURINGMACHINE

In de Appendix bij dit boek wordt een computerprogramma beschreven dat een Turingmachine simuleert. De input voor de simulatie bestaat uit de beschrijving van een Turingmachine $M = (Q, \Sigma, T, P, q_0, F)$ en van een inputstring $x \in T^*$. De TM beschrijving moet in een bepaalde voorgeschreven vorm worden gegeven, zoals aangegeven in de Appendix. Hier beschrijven we een *codering* van een TM, M met $e(M)$. Als het simulatieprogramma als input $e(M)$ en een string x, krijgt dan zal het programma M met input x simuleren. (We gaan uit van de veronderstelling dat steeds voldoende geheugencapaciteit beschikbaar is.) Als M met succes stopt na input x en dus een functie $f_M(x)$ berekent dan levert het simulatieprogramma ook $f_M(x)$ als resultaat. Als M daarentegen zonder succes stopt na input x, dan drukt het programma een mededeling met deze inhoud af. Zou echter M in een eindeloze herhalingslus terecht komen dan stopt ook het programma niet en wordt ook geen bericht afgedrukt. Helaas is het probleem van de mogelijke eindeloze lus onvermijdelijk en dit zullen we in dit hoofdstuk aantonen. Het is niet mogelijk een methode te bedenken waarmee het simulatieprogramma altijd kan vaststellen dat een willekeurige TM, M in een einde-

loze lus is blijven hangen en een bericht met die strekking kan dan ook niet worden afgedrukt.

Volgens de these van Church is elke functie die berekenbaar is met behulp van een Pascalprogramma, ook berekenbaar met behulp van een of andere Turingmachine. Er moet dus ook een TM bestaan die dezelfde acties uitvoert als ons simulatieprogramma. Ook deze TM kan steeds beschikken over voldoende geheugencapaciteit en er bestaat dus een TM, U die als input aanvaardt $e(M)$, de codering van een TM, $M = (Q, \Sigma, T, P, q\ , F)$ gevolgd door een scheidingssymbool, bijvoorbeeld *, gevolgd door een string $x \in T^*$. Gegeven de input $e(M)*x$ stopt U dan en slechts dan met succes als M met succes stopt na input x. De output van U is dan gelijk aan die van M, dus gelijk aan $f_M(x)$. U stopt zonder succes na input $e(M)*x$ desd als ook M zonder succes stopt na input x en U blijft hangen in een eindeloze lus na input $e(M)*x$ desd als M na input x in een eindeloze lus geraakt.



Figuur 3.1 Een universele Turingmachine

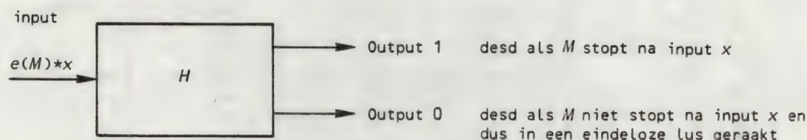
In figuur 3.1 is een en ander nog eens samengevat. Een dergelijke Turingmachine heet een *universele Turingmachine*. Hoe deze machine er precies uitziet zal afhangen van de aard van de codering van de inputstring. We gaan ervan uit dat de gebruikte codering 'zinnig' is. Hiermee bedoelen we dat (i) er een of andere effectieve procedure bestaat waarmee $e(M)$ uit M kan worden berekend, (ii) $e(M)$, M uniek bepaalt en dat de codering effectief decodeerbaar is, (iii) de codering geen onnodige opvulling met loze symbolen bevat. Een zinnige codering kan bijvoorbeeld worden afgeleid uit een Gödelnummering van M, (zie hoofdstuk 2).

Het probleem met de universele Turingmachine is dat deze andere Turingmachines eigenlijk te goed nabootst! Als M in een eindeloze lus raakt na input x, dan raakt ook de universele TM in een eindeloze lus

na input $e(M)*x$. De machine produceert dan geen output. Hoe kunnen we nu vaststellen wanneer we de universele TM kunnen afzetten? Deze kan zijn blijven hangen in een lus, maar het is ook mogelijk dat de machine zich midden in een berekening bevindt en uiteindelijk netjes zal stoppen. Veel bevredigender zou het zijn als we de universele TM zo konden aanpassen dat deze met gegeven input $e(M)*x$ ofwel de functie $f_M(x)$ berekent, ofwel het bericht geeft dat $f_M(x)$ onbepaald is. Daartoe zou de machine U moeten kunnen vaststellen of M is blijven hangen in een lus.

HET HALTINGPROBLEEM

Een algoritme is een procedure die altijd een waarneembaar resultaat oplevert. Formeler kunnen we een *algoritme* definiëren als een Turingmachine die uiteindelijk altijd stopt.



Figuur 3.2 Een onmogelijke Turingmachine

Als er een algoritme bestaat dat een bepaald probleem oplost, dan noemen we dat probleem *oplosbaar*, zo niet dan is het *onoplosbaar*. In hoofdstuk 2 hebben we bijvoorbeeld aangetoond dat optelling en vermenigvuldiging van gehele getallen oplosbare problemen zijn. Bestaat er nu ook een algoritme dat kan vaststellen of een Turingmachine al of niet zal stoppen na een bepaalde input; met andere woorden een algoritme dat het *haltingprobleem of stopprobleem voor Turingmachines* oplost? We formuleren de probleemstelling: gegeven een TM, $M = (Q, \Sigma, T, P, q_0, F)$ en een input $x \in T^*$, wordt gevraagd of M uiteindelijk zal stoppen.

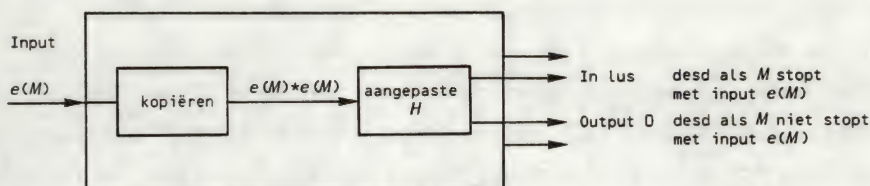
We zoeken dus naar een TM, H die zich gedraagt zoals weergegeven in figuur 3.2 en we zullen bewijzen dat zo'n machine onmogelijk kan bestaan! De daartoe te formuleren stelling is niet alleen een zeer belangrijke stelling, maar het bewijs is ook erg aardig en vrij subtiel.

Stelling 3.1

Het haltingprobleem voor Turingmachines is onoplosbaar.

Bewijs. We veronderstellen dat de gezochte TM, H bestaat en tonen aan dat deze veronderstelling tot een tegenspraak leidt. Construeer, uitgaande van de veronderstelling dat H bestaat, een tweede TM, H' en ga daarbij als volgt te werk. H' krijgt als input $e(M)$, een codering van een Turingmachine M, en kopieert deze codering in een string $e(M)*e(M)$. H' voert nu het programma van H uit met deze input, maar waar H als input 1 oplevert laten we H' in een eindeloze lus geraken. In figuur 3.3 is de werking van H' weergegeven.

Nu vragen we ons af wat er gebeurt als H' de string $e(H')$ als input krijgt. Op grond van bovenstaande constructie moet gelden dat H' in een eindeloze lus geraakt desd als H' stopt en dat H' stopt desd als H' in een eindeloze lus komt!



Figuur 3.3 De Turingmachine H'

Hier is duidelijk sprake van een tegenspraak en dus bestaat H' niet en H evenmin.

De onoplosbaarheid van het haltingprobleem voor Turingmachines is een opmerkelijk resultaat en dit heeft belangrijke gevolgen binnen de informatica. Omdat we een TM kunnen simuleren met een conventioneel computerprogramma dat gebruik maakt van een onbegrensd geheugen, moeten we concluderen dat we geen algoritme kunnen ontwerpen dat van een willekeurig programma met een willekeurige input kan vaststellen of dit zal stoppen of niet. Voor een bepaald programma kan weliswaar misschien wel worden vastgesteld dat het altijd zal stoppen, maar een algemene altijd werkende procedure hiervoor bestaat niet.

Gewoonlijk krijgt een programmeur een *specificatie* die precies omschrijft wat zijn programma moet bewerkstelligen. De opdracht kan bijvoorbeeld zijn een programma in zeg Pascal te schrijven dat voldoet aan de volgende specificatie:

INPUT: drie gehele getallen a, b en c

OUTPUT: de twee wortels van de vergelijking $ax^2 + bx + c = 0$ (voor zover deze bestaan).

Zoals zo vaak het geval is met specificaties in de praktijk, is ook deze specificatie onvolledig - een volledige specificatie zou ook aangeven wat de output moet zijn als de wortels niet bestaan, of als ze gelijk zijn, of als $a = 0$ enzovoort. Als dat allemaal is vastgesteld dan moet de programmeur een programma P schrijven dat aan specificatie S voldoet. Hij moet er zorg voor dragen dat het programma inderdaad het gestelde doel bereikt; in het ideale geval betekent dit dat hij voor alle mogelijke inputs waarvoor de specificatie S gedefinieerd is, moet bewijzen dat zijn programma P eindigt en de output genereert die volgens S bij de gegeven input behoort. Men noemt zo'n programma *totaal correct*.

Als we nu beschikken over een programmeertaal waarmee we een Turingmachine kunnen simuleren dan weten we dat er geen algemeen toepasbaar algoritme bestaat waarmee kan worden bewezen dat een willekeurig programma, gebruikmakend van onbeperkt geheugen, al dan niet zal stoppen. Zelfs als het geheugen, (zoals het geval is), eindig is dan nog is het probleem praktisch onoplosbaar, hoewel het theoretisch dan wel oplosbaar is. Elke machine van redelijke omvang kent zo veel toestanden dat het onmogelijk in de praktijk te testen is of een van die toestanden opnieuw wordt bereikt. Het ziet er naar uit dat hier sprake is van een ernstig probleem binnen de computerwetenschap, (en dat is inderdaad het geval!).

Het probleem kan op twee manieren worden geëlimineerd. Een mogelijkheid is, onze programmeertaal dermate te beperken dat wel een algoritme kan worden ontwikkeld dat kan vaststellen of een willekeurig programma in die taal zal stoppen of niet na ontvangst van een bepaalde input. Duidelijk is dat die beperkingen zo zwaar moeten zijn dat we in die taal de eenvoudige bewerkingen van een Turingmachine niet kunnen uitvoeren. Een betere benadering is wellicht het ontwikkelen

van algemeen toepasbare technieken die kunnen worden toegepast op programma's geschreven in de desbetreffende taal. Als deze technieken verstandig worden toegepast dan is het meestal mogelijk van een correct programma te bewijzen dat het zal eindigen. Hoe de techniek er precies uitziet hangt af van de beschouwde programmeertaal. In Pascal bijvoorbeeld kan een programma in een eindeloze lus raken als een while-statement niet meer wordt verlaten. Een while-statement ziet er zo uit:

while B do S

Hier bij is B een Boole'se expressie en S een statement of instructie. De expressie B wordt berekend en als deze logisch waar blijkt te zijn dan wordt S uitgevoerd en wordt het while-statement opnieuw ingevoerd. Alleen als B logisch onwaar wordt, kan de herhaling stoppen en is het while-statement voltooid. Het while-statement kan dus ook geschreven worden als:

if B then
begin
  S; while B do S
end

Veronderstel dat de Boole'se expressie van de vorm $x \subseteq c$ is, waarbij $\subseteq$ een totale ordeningsrelatie is, x een variabele en c een constante. De programmeur kan nu misschien bewijzen dat de lus niet eindeloos is, tenminste als hij (of zij) kan aantonen dat als bijvoorbeeld $x = x_i$ vóór de uitvoering van S, dat dan $x = x_{i+1}$ met $x_i \subset x_{i+1}$ ná de uitvoering van S. De waarde van x 'neemt dus toe' bij elke iteratie van het while-statement, maar het while-statement wordt alleen uitgevoerd zolang de waarde van x 'kleiner dan' c blijft. Onder voorbehoud dat de verzameling waarden die x kan aannemen c bevat en aftelbaar is, is men verzekerd van beëindiging.

Technieken voor het bewijzen van de correctheid van programma's berusten gewoonlijk op de mathematische logica. Zie bijvoorbeeld Dowsing et al. (1985), 'A first Course in Formal Logic and its Applications in Computer Science', of Dijkstra en Feijen (1984), 'Een methode van programmeren'. In deze boeken worden eenvoudige regels geformuleerd waarmee de programmeur de *partiële verificatie* van een correct

programma kan uitvoeren. Dit houdt in dat men aantoont dat als het programma eindigt dat het dan voldoet aan zijn specificatie. Eerlijkheidshalve moeten wij toevoegen dat van de meeste programma's die worden geschreven nooit formeel de partiële correctheid wordt aangetoond, laat staan de totale correctheid. Veel programma's bevatten dus fouten met alle mogelijke gevolgen van dien (besturingsfouten bij geleide projectielen, instortende bruggen, verkeerde managementsbeslissingen, enzovoort). Het is de verantwoordelijkheid van de programmeur om correcte code af te leveren en geen enkele code kan correct worden verondersteld als die correctheid niet bewezen is. Dergelijke bewijzen zijn moeilijk, vooral voor grote programma's en daarom beslist menig programmeur dat het grondig testen van de programmatuur met behulp van een uitgebreide testset voldoende moet zijn. Zolang er geen algemeen toepasbare methode bestaat die kan worden uitgevoerd met behulp van daartoe ontwikkelde software-gereedschappen, zal het testen ongetwijfeld in zwang blijven.

Het haltingprobleem voor Turingmachines is niet het enige onoplosbare probleem - er bestaat een aanzienlijk aantal van dergelijke problemen binnen verschillende toepassingsgebieden van de wiskunde en de informatica. Het bewijs van onoplosbaarheid kan vaak worden uitgevoerd door gebruik te maken van de bewezen onoplosbaarheid van het haltingprobleem. In dit hoofdstuk en in het volgende hoofdstuk geven we daarvan enkele voorbeelden. Om de behandeling te vereenvoudigen spreken we een formele notatie af.

Bij veel problemen is het mogelijk een overeenkomstig probleem te formuleren met als mogelijke uitkomsten 'ja' of 'nee'. Het probleem van het optellen van twee gehele getallen kan bijvoorbeeld worden geformuleerd als 'gegeven de gehele getallen x, y en z wordt gevraagd: geldt $z = x + y$?'. Problemen met ja/nee oplossingen worden *beslissingsproblemen* genoemd en, zoals gebruikelijk in de literatuur over berekenbaarheid, zullen we op dit soort problemen uitgebreid nader ingaan. We zullen beslissingsproblemen steeds in een standaardvorm formuleren. We beginnen met een omschrijving van het probleem, (mogelijk gevolgd door een verkorte naam). Vervolgens beschrijven we de gegevens bij het probleem en tenslotte formuleren we de gestelde vraag. Het optelprobleem ziet er dan zo uit:

Optellen van gehele getallen (SOM)

Gegeven: Drie gehele getallen x, y en z

Gevraagd: Geldt $z = x + y$?

Een tweede beslissingsprobleem dat we behandelden luidt:

Haltingprobleem (HP)

Gegeven: Een Turingmachine $M = (Q, \Sigma, T, P, q_0, F)$ en een string $x \in T^*$.

Gevraagd: Stopt M gegeven de input x?

Algemeen kan men stellen als Π een beslissingsprobleem is dan bestaat er een verzameling D_Π, het *domein* van Π, die alle mogelijke *instanties* of voorkomens van het probleem vertegenwoordigt. Bij sommige van deze voorkomens is het antwoord 'ja' en bij andere 'nee'. We kunnen D_Π dus verdelen in twee disjuncte deelverzamelingen Y_Π en N_Π, respectievelijk de *ja-instanties* en de *nee-instanties*. We noemen Π oplosbaar desd als er een TM, M_Π bestaat die, gegeven een (zinvolle) codering $e(I)$ van elke instantie $I \in D_\Pi$, altijd stopt met succes en met een output die aangeeft of wel of niet geldt dat $I \in Y_\Pi$. Zonder verlies aan algemeenheid kunnen we veronderstellen dat de output 1 is als $I \in Y_\Pi$ en dat de output 0 is indien dat niet het geval is. Als er echter geen Turingmachine bestaat die aan bovenstaande eisen voldoet, dan is het beslissingsprobleem *onoplosbaar*. SOM is een voorbeeld van een oplosbaar beslissingsprobleem en veel wat in hoofdstuk 2 behandeld is staat in direct verband met dit soort probleem. In dit hoofdstuk formuleerden we een onoplosbaar beslissingsprobleem (HP) en we zullen spoedig zien dat dit niet het enige is!

Beschouw twee beslissingsproblemen Π en Π'. We zeggen dat Π *reduceerbaar is* naar Π' (notatie $\Pi \sim \Pi'$) als een algoritme dat Π' oplost direct kan worden toegepast om Π op te lossen. Formeler: $\Pi \sim \Pi'$ desd als er een TM bestaat die als input $e(I)$ accepteert, dat wil zeggen een codering van een willekeurig voorkomen van Π, en die als output $e'(I')$ oplevert, een codering van een voorkomen I' van Π', zodanig dat $I \in Y_\Pi$ desd als $I' \in Y_{\Pi'}$. Steeds als we in dit boek het woord 'codering' gebruiken, bedoelen we een zinvolle codering en in dat geval is het gemakkelijk aan te tonen dat $\sim$ transitief is (zie oefening 3.4).

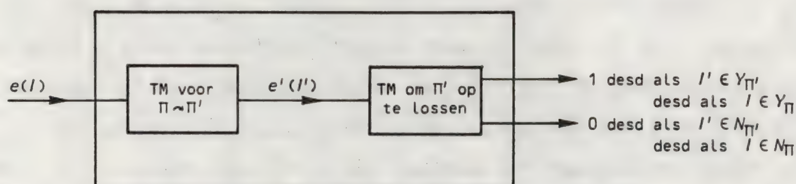
Ook de volgende belangrijke eigenschappen kunnen worden afgeleid.

Stelling 3.2

(1) Als $\Pi \sim \Pi'$ en Π' is oplosbaar dan is ook Π oplosbaar.

(2) Als $\Pi \sim \Pi'$ en Π is onoplosbaar dan is ook Π' onoplosbaar.

Bewijs. (1) Uit de definitie van $\sim$ volgt dat $I \in Y_\Pi$ desd als $I' \in Y_{\Pi'}$, terwijl er een TM bestaat die gegeven de input $e(I)$ voor elke $I \in D_\Pi$ als output $e'(I')$ levert. Omdat nu Π' oplosbaar is, bestaat er een andere TM die $e'(I')$ als input accepteert en vaststelt of al of niet $I' \in Y_{\Pi'}$. Het combineren van beide Turingmachines, zoals is weergegeven in figuur 3.4, leidt nu tot een algoritme voor het oplossen van Π.



Figuur 3.4 Een TM voor het oplossen van Π

(2) Als Π' oplosbaar was dan konden we uit (1) afleiden dat ook Π oplosbaar moet zijn en dit leidt tot een tegenspraak.

Op grond van dit resultaat en op grond van stelling 3.1 kunnen we nu vaststellen dat een beslissingsprobleem Π met HP $\sim \Pi$ onoplosbaar moet zijn. Deze conclusie zullen we gebruiken om te bewijzen dat de volgende problemen onoplosbaar zijn.

Het lege woord haltingprobleem (εHP)
Gegeven: Een TM, $M = (Q, \Sigma, T, P, q_0, F)$
Gevraagd: Stopt M gegeven de input ε?

Het uniforme haltingprobleem (UHP)

Gegeven: Een TM, $M = (Q,\Sigma,T,P,q_0,F)$

Gevraagd: Stopt M voor elke input $x \in T^*$?

HP $\rightsquigarrow$ εHP volgt uit de volgende reductie. Uit een instantie I van HP met een TM, $M = (Q,\Sigma,T,P,q_0,F)$ en een string $x \in T^*$ construeren we een instantie I' van HP met een TM, M'. M' wordt zodanig geconstrueerd dat eerst x op de tape wordt geschreven en vervolgens het gedrag van M wordt gesimuleerd. Duidelijk is dat M stopt bij een input $x \in T^*$ desd als M' stopt bij input ε. Dus geldt $I \in Y_{HP}$ desd als $I' \in Y_{\varepsilon HP}$.

HP $\rightsquigarrow$ UHP volgt uit een overeenkomstige maar iets subtielere reductie. Gegeven een instantie I van HP dat bestaat uit een TM, $M = (Q,\Sigma,T,P,q_0,F)$ en een string $x \in T^*$ construeren we een instantie I'' van UHP met een TM, M''. M'' wordt zodanig geconstrueerd dat eerst de input tape leeg wordt gemaakt, vervolgens x op de tape wordt geschreven en tenslotte het gedrag van M wordt gesimuleerd. M stopt dus na input $x \in T^*$ desd als M'' stopt voor alle inputs uit T^*. Dus geldt $I \in Y_{HP}$ desd als $I'' \in Y_{UHP}$. We hebben met deze beide reducties het volgende bewezen.

Stelling 3.3

(1) Het lege woord haltingprobleem (εHP) is onoplosbaar.

(2) Het uniforme haltingprobleem (UHP) is onoplosbaar.

Het probleem van het bewijs van de equivalentie van twee computerprogramma's houdt direct verband met het bewijzen van de correctheid van computerprogramma's. Twee programma's zijn equivalent als zij precies dezelfde taak uitvoeren. In termen van Turingmachines hebben we al gedefinieerd dat twee Turingmachines $M_1 = (Q_1,\Sigma_1,T_1,P_1,q_{01},F_1)$ en $M_2 = (Q_2,\Sigma_2,T_2,P_2,q_{02},F_2)$ equivalent zijn, desd als $f_{M_1} = f_{M_2}$. We definiëren nu:

Het equivalentieprobleem voor Turingmachines (ETM)

Gegeven: Twee Turingmachines $M_1 = (Q_1, \Sigma_1, T_1, P_1, q_{01}, F_1)$ en $M_2 = (Q_2, \Sigma_2, T_2, P_2, q_{02}, F_2)$ met hetzelfde inputalfabet T.

Gevraagd: Is M_1 equivalent met M_2?

Het antwoord is weinig verrassend.

Stelling 3.4

Het equivalentieprobleem voor Turingmachines is onoplosbaar.

Bewijs. We tonen aan dat UHP $\sim$ ETM. Zij een instantie I van UHP bepaald door een TM, $M = (Q, \Sigma, T, P, q_0, F)$. Uit M kunnen we een twee-tape machine M' construeren die haar input op de tweede tape kopieert en vervolgens het gedrag van M simuleert. Als en alleen als M stopt, maakt M' zijn eerste tape schoon, schrijft een 1 op positie één en stopt zelf ook. M' berekent dus

$$f_{M'}(x) = \begin{cases} 1 & \text{als } M \text{ stopt gegeven input } x \in T^* \\ \text{onbepaald} & \text{anders} \end{cases}$$

Veronderstel nu dat M'' de ééntape machine is die volgens stelling 2.4 uit M' kan worden geconstrueerd. Dan geldt dat $f_{M''} = f_{M'}$. Dus $I \in Y_{\text{UHP}}$ desd als $f_{M''}(x) = 1$ voor alle $x \in T^*$. Een TM, M_1 die $f_{M_1}(x) = 1$ berekent voor alle $x \in T^*$ is gemakkelijk te construeren. We definiëren M_1 eenvoudig als $M_1 = (\{q_0, q_1, q_2\}, T \cup \{\wedge\}, T, P, q_0, \{q_2\})$ met

$$P(q_0, a) = (q_0, 1, R) \quad \text{voor alle} \quad a \in T \cup \{\wedge\}$$

en $$P(q_1, a) = (q_2, \wedge, 0) \quad \text{voor alle} \quad a \in T \cup \{\wedge\}$$

De instantie I'' van ETM wordt nu bepaald door M'' en M_1 zoals hierboven geconstrueerd. $I \in Y_{\text{UHP}}$ desd $f_{M''} = f_{M_1}$ desd als $I' \in Y_{\text{ETM}}$. Omdat er een effectieve constructie van I' uit I bestaat, volgt dat UHP $\sim$ ETM en omdat UHP onoplosbaar is, is de stelling bewezen.

POST'S CORRESPONDENTIEPROBLEEM

Post's correspondentieprobleem (PCP) is ook een voorbeeld van een onoplosbaar probleem. Het bewijs dat HP$\sim$PCP is behoorlijk lastig, maar niettemin het bestuderen waard. De onoplosbaarheid van PCP is een belangrijk resultaat; het wordt in veel gevallen gebruikt waar het gaat om het bewijzen van de onoplosbaarheid van problemen uit de formele taaltheorie. In hoofdstuk 4 zullen wij hiertoe ook van dit resultaat gebruik maken.

In standaardvorm wordt PCP als volgt beschreven.

Post's correspondentieprobleem (PCP)

Gegeven: Een eindig alfabet T en twee n-tupels ($n > 0$) van strings in T^+, $\underset{\sim}{x} = (x_1, x_2, \ldots, x_n)$ en $\underset{\sim}{y} = (y_1, y_2, \ldots, y_n)$.

Gevraagd: Bestaat er een rij gehele getallen $i_1, i_2, \ldots, i_m$ ($m \geqslant 1$) zodanig dat $x_{i_1} x_{i_2} \ldots x_{i_m} = y_{i_1} y_{i_2} \ldots y_{i_m}$?

Een instantie van PCP met $T = \{0,1\}$, $\underset{\sim}{x} = (011, 11)$, $\underset{\sim}{y} = (0, 111)$ zit in Y_{PCP} omdat er een ja-oplossing bestaat, namelijk $x_1 x_2 x_3 = y_1 y_2 y_3 = 0111111$. Ook kan een voorbeeld worden gegeven van een voorkomen dat in Y_{PCP} zit, namelijk $T = \{0,1\}$, $\underset{\sim}{x} = (01, 100, 010)$ en $\underset{\sim}{y} = (010, 00, 100)$. (Zie ook oefening 3.6.) Dat men voorbeelden kan geven van voorkomens van PCP die wel en niet in Y_{PCP} zitten wil nog niet zeggen dat PCP oplosbaar is. Het is zelfs zo dat PCP, zoals hierboven geformuleerd onoplosbaar is. Bedenk dat alleen sprake is van oplosbaarheid als er een algoritme bestaat dat kan worden gebruikt voor het beantwoorden van de vraag of een *willekeurige* instantie $I \in D_{\text{PCP}}$ al dan niet in Y_{PCP} zit.

Een iets gewijzigde versie van PCP zal in het navolgende van belang blijken te zijn. Hij luidt als volgt:

Modificatie van Post's correspondentieprobleem (MPCP)

Gegeven: Een eindig alfabet T en twee n-tupels ($n > 0$) van strings in T^+, $\underset{\sim}{x} = (x_1, x_2, \ldots, x_n)$ en $\underset{\sim}{y} = (y_1, y_2, \ldots, y_n)$.

Gevraagd: Bestaat er een rij gehele getallen $i_1, i_2, \ldots, i_m$ ($m \geqslant 1$) met $i_1 = 1$ en $x_{i_1} x_{i_2} \ldots x_{i_m} = y_{i_1} y_{i_2} \ldots y_{i_m}$?

Het enige verschil tussen MPCP en PCP is dat nu de eerste string uit elke n-tupel de index 1 moet hebben. We tonen aan dat als MPCP onoplosbaar is dat dan ook PCP onoplosbaar is en we gebruiken daartoe de volgende stelling.

Stelling 3.5

MPCP $\rightsquigarrow$ PCP

Bewijs. Zij $I \in D_{\mathrm{MPCP}}$ bepaald door een alfabet T en twee n-tupels $\underset{\sim}{x} = (x_1, x_2, \ldots, x_n)$ en $\underset{\sim}{y} = (y_1, y_2, \ldots, y_n)$. Zonder verlies van algemeenheid mogen we veronderstellen dat T alleen symbolen heeft die in minstens één van de strings uit x of y voorkomen. Laten nu £ en \$ twee symbolen zijn die niet in T voorkomen en zij $T' = T \cup \{£, \$\}$. Definieer nu twee homomorfismen, $links: T^+ \to (T')^+$ en $rechts: T^+ \to (T')^+$ door middel van $links(a) = \$a$ en $rechts(a) = a\$$ voor alle $a \in T$. De functie *links* plaatst een \$ links en de functie *rechts* plaats een \$ rechts van elk symbool in een string.

Definieer vervolgens $\underset{\sim}{x}' = (x'_1, x'_2, \ldots, x'_n, £)$ en $\underset{\sim}{y}' = (y'_1, y'_2, \ldots, y'_n, \$£)$ als volgt:

$$x'_1 = concat(\$, rechts(x_1))$$
$$x'_i = rechts(x_i) \quad \text{voor} \quad 1 < i \leqslant n$$
$$y'_i = links(y_i) \quad \text{voor} \quad 1 < i \leqslant m$$

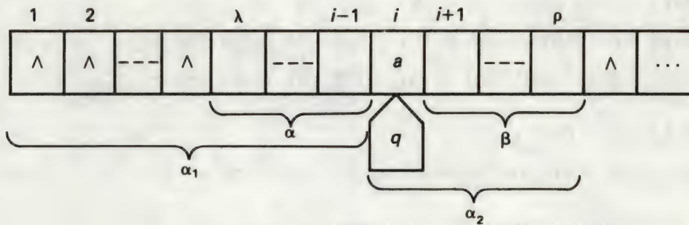
Als I' een instantie is van PCP bepaald door T', $\underset{\sim}{x}'$ en $\underset{\sim}{y}'$ dan is het niet moeilijk aan te tonen dat uit het feit dat $1, i_2, \ldots, i_m$ een oplossing is voor I, volgt dat $1, i_2, \ldots, i_m, n+1$ een oplossing is voor I'. Veronderstel nu dat $i_1, i_2, \ldots, i_{m'}$ een oplossing is voor I'. In dat geval moet $i_1 = 1$ (immers alle strings in $\underset{\sim}{y}'$ beginnen met \$ en alleen x_1 in $\underset{\sim}{x}$ begint met \$) en $i_{m'} = n+1$ (want alle strings in $\underset{\sim}{x}'$ behalve x'_{n+1} eindigen met \$ en geen enkel string in $\underset{\sim}{y}'$ eindigt met \$). Hieruit volgt dat $1, i_1, \ldots, i_{m'-1}$ een oplossing voor I moet zijn. Als we de tot nu toe gevonden resultaten combineren dan volgt dat $I' \in Y_{\mathrm{PCP}}$ desd als $I \in Y_{\mathrm{MPCP}}$. Omdat de constructie van I' uit I duidelijk effectief berekenbaar is, is hiermee het bewijs geleverd.

Stelling 3.6

HP ~ MPCP

Bewijs. Zij I een instantie van HP, bestaande uit een TM, $M = (Q,\Sigma,T,P,q_0,F)$ en een string $x \in T^*$. We zullen laten zien hoe op effectieve wijze een instantie I' van MPCP kan worden geconstrueerd zodanig dat $I \in Y_{HP}$ desd als $I' \in Y_{MPCP}$.

Op grond van stelling 2.5 mogen we veronderstellen dat M een eenzijdig onbegrensde tape bezit en een voor de hand liggende kleine aanpassing in stelling 2.1 laat de veronderstelling toe dat M nooit zonder succes stopt. We kunnen nu een configuratie $C = (q,i,\alpha,a,\beta)$ van de TM, M voorstellen door één enkele string $C \in (Q \cup \Sigma)^*$. C wordt voorgesteld door $\overline{C} = \alpha_1 q \alpha_2$ met $|\alpha_1| = i-1$, waarbij α_1 de string van symbolen links van de lees/schrijfkop voorstelt en $\alpha_2 = a\beta$ de string van symbolen op de locaties $i, i+1, i+2, \ldots$ tot een het meest rechtse niet-lege symbool, (zie figuur 3.5). Merk op dat α_1 niet noodzakelijk gelijk is aan α omdat α_1 kan beginnen met één of meer lege symbolen.



Figuur 3.5

De beginconfiguratie wordt voorgesteld door q_0x. Stel dat daarop volgende configuraties $\alpha_1 q_1 \beta_1, \alpha_2 q_2 \beta_2, \ldots, \alpha_m q_m \beta_m$ zijn met $q_m \in F$. We kunnen dan een voorkomen van MPCP construeren die een oplossing heeft met het prefix $\$q_0x\$\alpha_1q_1\beta_1\$\ldots\$\alpha_mq_m\beta_m\$$. Het symbool $\$ \notin Q \cup \Sigma$ introduceren we als nieuw symbool. Als M niet stopt na input x dan zal het voorkomen van MPCP dat we construeerden geen oplossing hebben.

Voor de constructie van de instantie I' van MPCP gaan we als volgt te werk. I' wordt gedefinieerd met het alfabet $Q \cup \Sigma \cup \{\$\}$, $x_1 = \$$

en $y_1 = \$q_0x\$$ vormen het eerste paar en de volgende corresponderende paren worden met behulp van de volgende regels geconstrueerd:

(1) voor elke $X \in \Sigma$ geldt dat X voorkomt in $\underset{\sim}{x}$ en in $\underset{\sim}{y}$;

(2) \$ komt zowel in $\underset{\sim}{x}$ als in $\underset{\sim}{y}$ voor;

(3) voor elke $q \in Q \backslash F$, $q' \in Q$ en $X, Y, Z \in \Sigma$ geldt

(a) ZqX komt voor in $\underset{\sim}{x}$ en $q'ZY$ komt voor in $\underset{\sim}{y}$ als $P(q,X) = (q', Y, L)$

(b) qX komt voor in $\underset{\sim}{x}$ en $q'Y$ komt voor in $\underset{\sim}{y}$ als $P(q,X) = (q', Y, 0)$

(c) qX komt voor in $\underset{\sim}{x}$ en Yq' komt voor in $\underset{\sim}{y}$ als $P(q,X) = (q', Y, R)$

(d) $Zq\$$ komt voor in $\underset{\sim}{x}$ en $q'ZY\$$ komt voor in $\underset{\sim}{y}$ als $P(q,\wedge) = (q', Y, L)$

(e) $q\$$ komt voor in $\underset{\sim}{x}$ en $q'Y\$$ komt voor in $\underset{\sim}{y}$ als $P(q,\wedge) = (q', Y, 0)$

(f) $q\$$ komt voor in $\underset{\sim}{x}$ en $Yq'\$$ komt voor in $\underset{\sim}{y}$ als $P(q,\wedge) = (q', Y, R)$

(4) voor elke $q \in F$, X, $Y \in \Sigma$

(a) XqY komt voor in $\underset{\sim}{x}$ en q komt voor in $\underset{\sim}{y}$

(b) $Xq\$$ komt voor in $\underset{\sim}{x}$ en $q\$$ komt voor in $\underset{\sim}{y}$

(c) $\$qY$ komt voor in $\underset{\sim}{x}$ en $\$q$ komt voor in $\underset{\sim}{y}$

(5) voor elke $q \in F$

$q\$\$$ komt voor in $\underset{\sim}{x}$ en \$ komt voor in $\underset{\sim}{y}$;

(6) behalve bovenstaande elementen komen geen andere elementen in $\underset{\sim}{x}$ en $\underset{\sim}{y}$ voor.

Veronderstel nu dat er uitgaande van de beginconfiguratie van M een toegelaten rij configuraties bestaat, weergegeven door $q_0x, \alpha_1q_1\beta_1, \alpha_2q_2\beta_2, \ldots, \alpha_kq_k\beta_k$ waarbij $q_0, q_1, \ldots, q_{k-1}$ geen eindtoestanden zijn. We beweren nu dat we elementen in $\underset{\sim}{y}$ kunnen vinden die te samen de string $\$q_0x\$\alpha_1q_1\beta_1\$\ldots\$\alpha_kq_k\beta_k\$$ vormen. Evenzo vormen de corresponderende elementen van x de string $\$q_0x\$\alpha_1q_1\beta_1\$\ldots\$\alpha_{k-1}q_{k-1}\beta_{k-1}\$$. Deze bewering kunnen we bewijzen door inductie naar k. Voor $k = 0$ geldt de bewering omdat we allereerst het paar $(\$, \$q_0x\$)$ moeten kiezen. Veronderstel nu dat de bewering geldt voor alle $i < k$ en in het

bijzonder voor $i = k-1$. We kunnen dan dus elementen in $\underset{\sim}{y}$ vinden die te samen de string $\$q_0x\$\alpha_1q_1\beta_1\$\ldots\$\alpha_{k-1}q_{k-1}\beta_{k-1}\$$ vormen en zodanig dat de corresponderende elementen in $\underset{\sim}{x}$ de string $\$q_0x\$\alpha_1q_1\beta_1\$\ldots$ $\ldots\$\alpha_{k-2}q_{k-2}\beta_{k-2}\$$ vormen. We willen nu de string $\alpha_{k-1}q_{k-1}\beta_{k-1}\$$ aan $\underset{\sim}{x}$ toevoegen. Alle symbolen in deze string behalve q_{k-1} kunnen we verkrijgen door gebruik te maken van de paren die we met de regels (1) en (2) construeerden. Maar omdat $q_{k-1} \notin F$ kunnen we q_{k-1} alleen verkrijgen uit een paar dat we construeren met behulp van regel (3). Dit laatste paar representeert een stap uitgevoerd door de Turingmachine en op deze manier corresponderen met de elementen van $\underset{\sim}{x}$ die te samen $\alpha_{k-1}q_{k-1}\beta_{k-1}\$$ vormen juist die elementen van $\underset{\sim}{y}$ die samen $\alpha_kq_k\beta_k\$$ vormen. De veronderstelling geldt dus ook voor $i = k$ en hiermee is het inductiebewijs voltooid. In feite hebben we zelfs een iets stringenter resultaat verkregen, namelijk dat elk vergelijkingsproces dat begint met $(\$,\$q_0x\$)$ strings zal genereren van de zo juist beschreven vorm.

Veronderstel nu dat we uiteindelijk een situatie bereiken waarin we een string $\$q_0x\$\alpha_1q_1\beta_1\$\ldots\$\alpha_{m-1}q_{m-1}\beta_{m-1}\$$ uit de symbolen van $\underset{\sim}{x}$ hebben geconstrueerd en een string $\$q_0x\$\alpha_1q_1\beta_1\$\ldots\$\alpha_mq_m\beta_m\$$ uit de overeenkomstige symbolen van $\underset{\sim}{y}$. Stel verder dat $q_m \in F$. We kunnen nu door gebruik te maken van de symbolenparen die worden geconstrueerd met de regels (4) en (5) gemakkelijk een oplossing voor MPCP verkrijgen die de string $\$q_0x\$\alpha_1q_1\beta_1\$\ldots\$\alpha_mq_m\beta_m\$$ als prefix heeft. Als M met input x dus een eindtoestand bereikt dan heeft het voorkomen I' van MPCP een oplossing. Als echter M geen eindtoestand bereikt dan zal de string van symbolen uit $\underset{\sim}{y}$ altijd langer zijn dan de string met corresponderende symbolen uit $\underset{\sim}{x}$ en kan er dus geen oplossing voor dit voorkomen van MPCP bestaan. We hebben hiermee aangetoond dat $I \in Y_{\text{HP}}$ dan en slechts dan als $I' \in Y_{\text{MPCP}}$ en omdat de constructie van I' uit I effectief berekenbaar is, is hiermee de stelling bewezen.

De volgende stelling volgt onmiddellijk uit de twee voorgaande.

Stelling 3.7

PCP is een onoplosbaar probleem.

ANDERE ONOPLOSBARE PROBLEMEN

Onoplosbare problemen komen voor in veel verschillende takken van de wiskunde. We geven hier slechts drie voorbeelden. Lezers wier kennis van de wiskunde tekort schiet kunnen deze paragraaf gerust overslaan. Later in dit boek wordt niet op de hier verkregen resultaten teruggegrepen; we nemen ze alleen maar volledigheidshalve op voor geïnteresseerden in deze materie. In het volgende hoofdstuk worden weer voorbeelden gegeven van onoplosbare problemen die wat gemakkelijker te begrijpen zijn.

Ons eerste voorbeeld staat bekend als *Hilbert's tiende probleem.* Beschouw een polynoom $p(x_1, x_2, \ldots, x_n)$ met variabelen $x_1, x_2, \ldots, x_n$ en geheeltallige coëfficiënten. De vergelijking

$$p(x_1, x_2, \ldots, x_n) = 0$$

heet een *diophantische vergelijking* als men geheeltallige oplossingen eist. Natuurlijk hebben dergelijke vergelijkingen vaak geen oplossingen; zo heeft bijvoorbeeld $x^2-2 = 0$ geen geheeltallige oplossing. In het jaar 1900 hield de wiskundige Hilbert een beroemde lezing, waarin hij een aantal problemen formuleerde die naar zijn mening door wiskundigen in de twintigste eeuw zouden moeten worden bestudeerd. Zijn tiende probleem was of er al dan niet een effectieve procedure bestaat om vast te stellen of een bepaalde diophantische vergelijking een oplossing heeft. Pas in 1972 toonde Matijacevič aan dat het bepalen van de existentie van oplossingen van diophantische vergelijkingen ook een onoplosbaar probleem is. Het werk van Matijacevič bouwde voort op eerdere studies op dit gebied door M. Davis, J. Robinson en H. Putman. Zie bijvoorbeeld Bell & Machover (1977) voor een gedetailleerde behandeling.

Binnen de mathematische logica komen ook een aantal onoplosbare problemen voor. Het meest fundamentele probleem, dat ook in elk leerboek over dit onderwerp wordt behandeld, is dat van de validiteit van de eerste orde predicatenlogica. Het is dan en slechts dan mogelijk een axiomastelsel voor de predicatenlogica te formuleren waarmee elke bewering uit deze logica kan worden bewezen, als deze logica *valide* is, dat wil zeggen waar onder iedere mogelijke interpretatie. In Church (1936b) wordt bewezen dat bewijsbaarheid (en dientengevolge validiteit) in de predicatenrekening onoplosbaar is. Dit is waarschijnlijk het meest

fundamentele onoplosbaarheidsprobleem in de gehele wiskunde. Binnen de computerwetenschap betekent het dat automatisch bewijzen van stellingen slechts ten dele mogelijk is. In Dowsing, Rayward-Smith & Walter (1985) wordt hierop nader ingegaan. Het probleem van het automatisch bewijzen van stellingen heeft directe invloed op de mogelijkheden bij het gebruik van de logica als programmeertaal (zie Kowalski 1979).

Ons laatste voorbeeld stamt uit de groepentheorie. Stel G is een groep met eenheidselement e en met elementen $x_1, x_2, \ldots$. Een *woord* in G is iedere string die uit deze elementen wordt geconstrueerd door gebruik te maken van de groepoperator * en inversen. Bijvoorbeeld $x_1 * x_1 * x_2^{-1} * x_3 * x_1^{-1} * x_4$ is een woord in G. Daar G gesloten is onder * en de inverse, stelt elk woord een element in G voor. Het volgende probleem is onoplosbaar.

Woordprobleem voor groepen (WPG)
Gegeven: Een groep G en een woord w in G.
Gevraagd: Is $w = e$, het identiteitselement van G?

Als we het probleem beperken tot eindige groepen dan is het wel oplosbaar.

OEFENINGEN

1. Laat zien hoe een TM kan worden gecodeerd als een string in $\{0,1\}^*$. Geef duidelijk aan welke veronderstellingen u maakt.

2. (a) Ontwerp een TM, M die het resultaat 1 oplevert voor alle inputs uit $\{0,1\}^*\{00\}$, maar die 0 oplevert voor alle andere inputs in $\{0,1\}^*$.
   (b) Definieer het probleem *geheeltallige deling door vier* in de standaard notatie. Gebruik uw antwoord op deel (a) om te laten zien dat dit probleem oplosbaar is.

3. Als Π een beslissingsprobleem is dan is het complement van Π, Π^c gedefinieerd als Π met ontkenning van de vraag. Toon aan dat $\Pi^c \sim \Pi$ en dat Π dus oplosbaar is desd als Π^c oplosbaar is.

4. Bewijs dat $\sim$ reflexief en transitief is maar niet symmetrisch.

5. Bewijs dat het volgende probleem oplosbaar is.

   Houdt soms halt (SH)

   Gegeven: Een TM, $M = (Q, \Sigma, T, P, q_0, F)$.

   Gevraagd: Bestaat er een $x \in T^*$ zodanig dat M stopt na input x?

6. Laat zien dat $T = \{0,1\}$, $\underset{\sim}{x} = (01,100,010)$, $\underset{\sim}{y} = (010,00,100)$ een voorkomen is van PCP binnen N_{PCP}.

7. Is PCP oplosbaar als het alfabet wordt beperkt tot slechts één symbool? En hoe zit het bij twee symbolen?

8. Construeer een voorkomen van MPCP dat overeenkomt met het voorkomen van HP met de TM die de functie kop berekent met de input 011 (zie figuur 2.4c).

9. Gebruik uw antwoord op vraag 8 om een voorkomen van PCP te construeren dat overeenkomt met het gegeven voorkomen van HP.

4

Formele Talen

... een belangrijk deelgebied van de informatica

J.E. HOPCROFT & J.D. ULLMAN

Formele Talen en hun Relatie tot Automaten

TURINGMACHINES ALS HERKENNERS

Een TM die een beslissingsprobleem Π oplost, gedraagt zich in principe als een herkenner; de machine herkent of een gegeven input $e(I)$ al of niet voorkomt in de een of andere taal $L_{\Pi,e} = \{e(I) | I \in Y_{\Pi}\}$. In dit hoofdstuk zullen we ons bezighouden met de talen die voor Turingmachines herkenbaar zijn en we zullen hun eigenschappen bestuderen.

Een taal $L \subseteq T^*$ heet *recursief* dan en slechts dan als er een Turingmachine bestaat met inputalfabet T, die de *karakteristieke functie* van L berekent,

$$\chi_L(x) = \begin{cases} 1 & \text{als} \quad x \in L \\ 0 & \text{als} \quad x \in T^* \setminus L. \end{cases}$$

Merk op dat een ongedefinieerde output voor enige input $x \in T^*$ hier niet is toegelaten. De TM moet dus altijd stoppen met succes en daarom kan de TM voor elke $x \in T^*$ beslissen of $x \in L$.

Recursieve talen kunnen ook op een andere maar equivalente wijze worden gekarakteriseerd. We zeggen dat een string $x \in T^*$ wordt *geaccepteerd* door een TM, $M = (Q, \Sigma, T, P, q \ , F)$ desd als M stopt met succes na input x. Een string $x \in T^*$ wordt *geweigerd* door de TM desd als M stopt zonder succes na input x. Als M in een eindeloze lus geraakt na input x dan kunnen we noch beweren dat M x accepteert, noch dat M x weigert.

Stelling 4.1

$L \subseteq T^*$ is recursief desd als er een TM, $M = (Q, \Sigma, T, P, q_0, F)$ bestaat zodanig dat M alle $x \in L$ accepteert en alle $x \in T^* \backslash L$ weigert.

Bewijs. ⇒: L is recursief en we mogen dus veronderstellen dat er een TM, M' bestaat die χ_L berekent. Op grond van stelling 2.5 mogen we aannemen dat M' een TM is met een eenzijdig onbegrensde tape. M' kan niet stoppen zonder succes omdat χ_L een totale functie is. We kunnen nu uit M' een TM, M construeren, eveneens met een eenzijdig onbegrensde tape. M bootst de werking van M' na, maar plaatst een merking in locatie 1 op de tape, zodanig dat als M' tijdens een berekening een symbool $X \in \Sigma$ op locatie 1 heeft staan, M daar het paar $(X,*)$ heeft staan. Het symbool * is nieuw en heeft geen verdere invloed op de berekening; het is louter een merking van locatie 1 op de tape. Op het moment dat M' zou stoppen, gebruikt M deze merking om (zonodig) de lees/schrijfkop naar locatie 1 te bewegen. Als daar het paar $(1,*)$ wordt gelezen dan komt M in zijn unieke eindtoestand q_f. Als $(0,*)$ wordt gelezen dan is er geen volgende actie gedefinieerd.

⇐: We gebruiken opnieuw stelling 2.5 en veronderstellen zonder verlies aan algemeenheid dat M een TM is met eenzijdig onbegrensde tape. We veronderstellen verder dat aan $M = (Q, \Sigma, T, P, q_0, F)$ een extra toestand q_d is toegevoegd, zodanig dat de waarde van $P(q,a)$ in alle gevallen waarin deze ongedefinieerd was nu gelijk wordt aan $(q_d, a, 0)$. De aangepaste TM kan dus alleen een inputstring verwerpen in toestand q_d. De constructie van een eenzijdig begrensde TM, M' ter berekening van χ_L is nu eenvoudig. M' bootst de werking van M na, maar merkt weer locatie 1. Als de simulatie van de aangepaste machine M leidt tot het bereiken van toestand q_d, dan zoekt M' locatie 1 op en schrijft daar een 0 gevolgd door het symbool $\wedge$ op locatie 1. Vervolgens stopt M'. In alle andere gevallen komt de aangepaste M in een toestand uit F. Ook nu gaat M' naar locatie 1, maar schrijft daar nu een 1, gevolgd door een $\wedge$ op locatie 2 en stopt.

Helaas bestaat de mogelijkheid dat een TM in een eindeloze lus geraakt na een bepaalde input. In hoofdstuk 3 toonden we aan dat het haltingprobleem onoplosbaar is en dat we dus op geen enkele manier kunnen vaststellen of een willekeurige TM, M al of niet zal stoppen na

een willekeurige input $x \in T^*$. Onze definitie van recursiviteit laat niet toe dat een TM in een lus terecht komt en dus kunnen niet alle door TM's herkenbare talen er mee worden gekarakteriseerd. We definiëren daarom de verzameling van *recursief enumereerbare* (r.e.) *talen* in T^* als die talen $L \subseteq T^*$ waarvoor een TM bestaat die de *partiële karakteristieke functie* van L berekent:

$$\chi'_L(x) = \begin{cases} 1 & \text{als } x \in L \\ \text{onbepaald} & \text{als } x \in T^* \setminus L \end{cases}$$

Nu kan de TM voor iedere $x \in L$ bevestigen dat x in L voorkomt, maar voor een willekeurige $x \in T^*$ kan het voorkomen dat er helemaal geen output wordt geproduceerd. We kunnen nu eenvoudig de volgende stelling bewijzen.

Stelling 4.2

Elke recursieve taal is recursief enumereerbaar.

Bewijs. Pas eenvoudig de TM, M die χ_L berekent zo aan dat deze in een eindeloze lus geraakt in plaats van te stoppen met output 0.

De omkering van deze stelling is niet waar; wellicht weinig verrassend na lezing van het vorige hoofdstuk. Toch is hier sprake van een belangrijk resultaat omdat hieruit blijkt dat men in het algemeen niet alleen niet vast kan stellen of een TM zich in een eindeloze lus bevindt, maar dat in zekere zin het terecht komen in een eindeloze lus onvermijdelijk is. Om te laten zien dat er r.e. talen zijn die niet recursief zijn zullen we eerst wat eigenschappen van de talen die we definieerden afleiden.

Stelling 4.3

(1) Recursieve talen zijn gesloten onder vereniging, doorsnede en concatenatie van verzamelingen. Dat wil zeggen: als $L_1, L_2 \subseteq T^*$ recursief zijn dan geldt dit ook voor $L_1 \in L_2$, $L_1 \in L_2$ en $L_1 L_2$.

(2) Recursief enumereerbare talen zijn gesloten onder vereniging, doorsnede en concatenatie.

Bewijs. We geven een schets van het bewijs dat recursieve talen gesloten zijn onder vereniging. De andere bewijzen verlopen analoog.

Veronderstel dat χ_{L_1}, χ_{L_2} berekend worden door respectievelijk de TM's M_1 en M_2. Beschouw nu een als volgt te construeren 3-tape machine M. Na elke input $x \in T^*$ op zijn eerste tape, kopieert M deze input naar tape 2 en 3. Op tape 2 bootst M de werking van M_1 na en op tape 3 die van M_2. Als uit de simulatie blijkt dat zowel $x \in L_1$ als $x \in L_2$ dan maakt M zijn inputtape op locatie 1 na leeg. Op locatie 1 schrijft M het symbool 1. In alle overige gevallen maakt M zijn inputtape eveneens schoon, maar schrijft een 0 op locatie 1.

Stelling 4.4

Recursieve talen zijn gesloten onder complement. Dat wil zeggen als $L \subseteq T^*$ recursief is dan is ook $\bar{L} = T^* \backslash L$ dat.

Bewijs. Als M L herkent dan herkent $\bar{M}$ $\bar{L}$. $\bar{M}$ wordt uit M geconstrueerd door elke output 1 door een output 0 te vervangen en omgekeerd.

We kunnen nu alle strings in T^* aftellen zodat x_i de i-de string voorstelt, (zie oefening 4 uit hoofdstuk 1). We kunnen ook een verzameling van TM's aftellen die r.e. talen in T^* herkennen, zodanig dat elke r.e. taal in T^* door minstens één machine wordt herkend, (oefening 4.1). Stel L_i is de i-de taal en deze wordt herkend door de TM, M_i. Dus geldt $x_i \in L_i$ desd als $f_{M_i}(x_i) = 1$. Laten we nu de taal

$$L_d = \{x_i \mid x_i \in L_i\}$$

eens bekijken.

Stelling 4.5

L_d is een r.e. taal, maar $\bar{L}_d$ is dat niet.

Bewijs (Schets). $\bar{L}_d$ wordt herkend door een machine M_d die als volgt werkt. Als M_d een input $x \in T^*$ krijgt dan begint M_d eerst met de aftelling $x_1, x_2, \ldots$ van T^* om de index i met $x_i = x$ te bepalen. M_d genereert dan de codering van een machine M_i en geeft de besturing over aan een universele TM die M_i met input x_i simuleert. Dus geldt dat $x = x_i$ met $f_{M_i}(x_i) = 1$ desd als $f_{M_d}(x) = 1$. Dus $f_{M_d} = {}'(L_d)$ dus is L_d r.e.

Als ook $\bar{L}_d = T^*\backslash L_d$ r.e. is dan moet deze taal herkend worden door een of andere TM, $\bar{M}_d$ met inputalfabet T. Bij enumeratie van deze TM's volgt $\bar{M}_d = M_k$ voor een of andere k. Dus geldt $x_k \in \bar{L}_d$ desd als x_k herkend wordt door M_k. Uit de definitie van $\bar{L}_d$ volgt echter dan $x_k \in \bar{L}_d$ desd als x_k niet wordt herkend door M_k. Uit deze tegenspraak volgt dat $\bar{L}_d$ niet r.e. kan zijn.

Hiermee hebben we laten zien dat recursieve talen gesloten zijn onder complement, maar dat dit niet geldt voor r.e. talen. We bewezen dus:

Stelling 4.6

Niet elke r.e. taal is ook recursief.

NIET-DETERMINISTISCHE TURINGMACHINES

Toen we in hoofdstuk 2 het begrip Turingmachine definieerden, definieerden we een programma als een partiële functie van $(Q\backslash F) \times \Sigma \rightarrow Q \times \Sigma \times \{L, R, 0\}$. Door te eisen dat een programma een functie is, zorgen we ervoor dat $P(q,a)$, $q \in Q\backslash F$, $a \in \Sigma$, steeds hoogstens één waarde kan hebben. We kunnen ook een minder strenge eis stellen en als we toelaten dat de machine meer dan één mogelijkheid heeft voor zijn volgende stap dan noemen we de machine niet-deterministisch of nondeterministisch. Een niet-deterministische Turingmachine (NDTM) wordt formeel gedefinieerd op dezelfde wijze als een deterministische, behalve dat nu het programma een functie $(Q\backslash F) \times \Sigma \rightarrow 2^{Q \times \Sigma \times \{L, R, 0\}}$ wordt. Dus gegeven $q \in Q\backslash F$ is $P(q,a)$ nu een verzameling van mogelijke volgende acties. (Deze verzameling kan ook leeg zijn.)

We zeggen dat een string $x \in T^*$ geaccepteerd wordt door een NDTM, $M = (Q, \Sigma, T, P, q_0, F)$ desd als er een of andere rij keuzes van acties bestaat waardoor M na input x uiteindelijk in een eindtoestand terecht komt. De verzameling van al dergelijke strings wordt weergegeven door $T(M)$, de *taal geaccepteerd* door M. Een deterministische TM kan worden gezien als een NDTM zodanig dat voor $q \in Q \backslash F$, $a \in \Sigma$, $P(q, a)$ ofwel leeg is, ofwel een singletonverzameling. Het is vervolgens niet moeilijk, gebruikmakend van eenzelfde redenering als in het bewijs van stelling 4.1 om de volgende stelling te bewijzen.

Stelling 4.7

L wordt geaccepteerd door een deterministische TM desd als L r.e. is.

Als we weten dat deterministische TM's iedere effectief berekenbare functie kunnen berekenen, dan vertegenwoordigen r.e. talen de verzameling van 'effectief herkenbare' talen. Men zou dus niet verwachten dat een uitbreiding naar nondeterminisme het mogelijk maakt voor een TM om niet r.e. talen te accepteren en dat is dan ook niet zo. Nondeterminisme heeft eigenlijk alleen betekenis als het gaat het het vergroten van de rekensnelheid, (zie hoofdstuk 6).

Stelling 4.8

L wordt geaccepteerd door een nondeterministische TM desd als L r.e. is.

Bewijs. $\Leftarrow$: een onmiddellijk gevolg van stelling 4.7 omdat we een deterministische TM kunnen beschouwen als een beperkte NDTM.

$\Rightarrow$: we moeten aantonen dat L wordt geaccepteerd door een deterministische TM als L wordt geaccepteerd door een NDTM. Veronderstel dat $M = (Q, \Sigma, T, P, q\ , F)$ een nondeterministische machine is zodanig dat $L = L(M)$. We zullen nu de constructie beschrijven van een deterministische TM, M' met $M' = T(M')$. Beschouw de verzamelingen $P(q, a)$,

$q \in Q$ en $a \in \Sigma$ en zij m de maximale cardinaliteit van deze verzamelingen. Als $m = 1$ dan is M deterministisch en hebben we niets te bewijzen. We veronderstellen dus $m > 1$. Iedere eindige rij van keuzes van acties voor M kan worden gecodeerd als een eindige rij getallen uit de getallen $1, 2, \ldots, m$. Niet iedere rij zal een mogelijke rij acties voorstellen omdat $\#P(q,a)$ kleiner dan m kan zijn voor een $q \in Q$ en $a \in \Sigma$.

M' is een 3-tape TM. Op de eerste tape schrijven we de input $x \in T^*$. Op de tweede tape genereert M' alle mogelijke rijen uit de getallen $1, 2, \ldots, m$ in *lexicografische* volgorde. Dat wil zeggen beginnend bij de kortste rijen en van klein naar groot. Bij elke gegenereerde rij wordt de input naar tape 3 gekopieerd en op tape 3 simuleert M' de machine M, waarbij de inhoud van tape 2 wordt gebruikt om de juiste actie te kiezen. Als er voor een bepaald getal in de rij geen bijbehorende actie bestaat dan stopt M' met het simuleren van M, tape 3 wordt leeg gemaakt, de volgende rij wordt op tape 2 gegenereerd en de simulatie van M wordt opnieuw begonnen met een nieuwe kopie van de input x op tape 3. Als er een rij keuzes van acties bestaat die leidt tot acceptatie van x door M dan komt deze rij uiteindelijk ook voor op tape 2 van M'. Als M' die rij bereikt dan wordt M gesimuleerd en wordt ook door M' de input x geaccepteerd. Als een dergelijke rij niet bestaat dan genereert M' steeds langere rijen van mogelijke acties en geraakt op deze wijze in een eeuwige lus.

FRASESTRUCTUUR GRAMMATICA'S

Frasestructuur grammatica's bestaat uit een eindige verzameling van afleidingsregels (produktieregels genaamd) waarmee strings kunnen worden gegenereerd. Deze strings worden gevormd met behulp van een of ander alfabet van terminale symbolen T. Verder gebruikt de grammatica ook een alfabet N van niet-terminale symbolen. De alfabetten T en N hebben geen symbolen gemeen. De afleidingsregels zijn alle van de vorm $\alpha \to \beta$ met $\alpha \in (N \cup T)^+$ en $\beta \in (N \cup T)^*$. Als we beginnen met een voorgeschreven startsymbool S uit N dan kan een string uit T^* worden gegenereerd door herhaaldelijk substrings die overeenkomen met een

linkerlid van een produktieregel te vervangen door het bijbehorende rechterlid.

Een eenvoudige grammatica met $N = \{S,A,B\}$ en $T = \{a,b\}$ kan bijvoorbeeld de volgende produktieregels hebben.

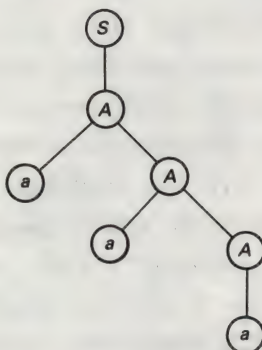
$$S \to A$$
$$S \to B$$
$$A \to aA$$
$$A \to a$$
$$B \to bB$$
$$B \to b$$

Uit S kunnen we A of B afleiden. Leiden we A af dan kunnen we vervolgens in één stap de string a afleiden en in twee stappen de string aa (dit laatste kan door uit A eerst aA af te leiden en vervolgens A door a te vervangen). In k stappen kunnen we zo een string van k a's genereren en op dezelfde manier kunnen we uit B een string van k b's afleiden in k stappen.

Als uit een string α een string β kan worden afgeleid door een van de produktieregels uit de grammatica toe te passen dan schrijven we $\alpha \Rightarrow \beta$. Als geldt dat $\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{n-1} \Rightarrow \alpha_n$ dan korten we dit af tot $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ of nog korter tot $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$. Uitgaande van de in het voorgaande gegeven grammatica krijgen we dan dat

$$S \Rightarrow A \Rightarrow aA \Rightarrow aaA \Rightarrow aaa$$

een toegelaten afleiding van a^3 uit S is. Zo'n afleiding kunnen we ook weergeven met behulp van een afleidingsboom zoals in figuur 4.1.



Figuur 4.1

We zullen nu de gebruikte begrippen formeel beschrijven.

Een *frasestructuur grammatica* (PSG) is een 4-tupel $G = (N, T, P, S)$ met

(1) N is een eindige verzameling van *nonterminale symbolen*. het is gebruikelijk de elementen van N weer te geven door hoofdletters (mogelijk met subscripts).

(2) T is een eindige verzameling van *terminale symbolen*, zodanig dat $N \cap T = \emptyset$. Gewoonlijk worden de elementen van T weergegeven door kleine letters (mogelijk met subscript en meestal uit het begin van het alfabet).

(3) P is een eindige verzameling *produktieregels* van de vorm $\alpha \to \beta$, waarbij α het linker lid van de regel zodanig is dat $\alpha \in (N \cup T)^+$ terwijl β, het rechterlid zodanig is dat $\beta \in (N \cup T)^*$.

(4) $S \in N$ is een speciaal symbool, aangewezen als *startsymbool* van de grammatica.

De PSG uit ons voorbeeld kan dus formeel worden beschreven door het quadrupel $G_1 = (\{S, A, B\}, \{a, b\}, P, S)$ waarbij P de volgende verzameling produktieregels voorstelt

$$\{S \to A, S \to B, A \to aA, A \to a, B \to bB, B \to b\}.$$

Meestal gebruiken we niet de formele verzamelingtheoretische notatie, maar schrijven we de produktieregels uit als een lijst, waarbij we het teken | gebruiken in de betekenis van 'of'. Bovenstaande produktieregels worden dan

$$S \to A \mid B$$
$$A \to aA \mid a$$
$$B \to bB \mid B$$

Door het gebruik van grote en kleine letters voor nondeterminalen en terminalen hoeven we alleen maar de produktieregels en het startsymbool te specificeren om de grammatica te beschrijven. Als we vast afspreken dat we het startsymbool met S weergeven dan hoeven we ook dat niet verder te specificeren.

Beschouw nu de PSG G_2 met produktieregels

$$\begin{aligned}
S &\rightarrow aSBC \mid aBC \\
CB &\rightarrow BC \\
aB &\rightarrow ab \\
bB &\rightarrow bb \\
bC &\rightarrow bc \\
cC &\rightarrow cc
\end{aligned}$$

Dit is een voorbeeld waarin de linker leden van de produktieregels niet allen uit enkele nonterminalen bestaan. Men kan aantonen dat uit S elke string van de vorm $a^n b^n c^n$ $(n \geqslant 1)$ kan worden geproduceerd. Bijvoorbeeld

$$\begin{aligned}
S &\Rightarrow aSBC \\
&\Rightarrow aaBCBC \quad (\text{gebruik } S \rightarrow aBC) \\
&\Rightarrow aabCBC \quad (\text{gebruik } aB \rightarrow ab) \\
&\Rightarrow aabBCC \quad (\text{gebruik } CB \rightarrow BC) \\
&\Rightarrow aabbCC \quad (\text{gebruik } bB \rightarrow bb) \\
&\Rightarrow aabbcC \quad (\text{gebruik } bC \rightarrow bc) \\
&\Rightarrow aabbcc \quad (\text{gebruik } cC \rightarrow cc)
\end{aligned}$$

is een toegelaten afleiding van $a^2b^2c^2$.

Hiermee moet het intuïtieve begrip 'afleiding' duidelijk zijn. We zullen nu gebruik maken van de formele definitie van een grammatica om precies te formuleren wat het betekent als een string afleidbaar is binnen een bepaalde grammatica. Zij $G = (N, T, P, S)$ een willekeurige PSG en zij $\gamma_1\alpha\gamma_2 \in (N \cup T)^+$ een string van terminalen en nonterminalen met lengte groter of gelijk 1. Als $\alpha \rightarrow \beta$ een produktieregel in P is dan kan α in $\gamma_1\alpha\gamma_2$ vervangen worden door β zodat we $\gamma_1\beta\gamma_2$ krijgen. We schrijven dan

$$\gamma_1\alpha\gamma_2 \underset{G}{\Rightarrow} \gamma_1\beta\gamma_2$$

[lees: $\gamma_1\alpha\gamma_2$ *genereert* $\gamma_1\beta\gamma_2$ of $\gamma_1\beta\gamma_2$ *is afgeleid uit* $\gamma_1\alpha\gamma_2$].

Als $\alpha_1, \alpha_2, \ldots, \alpha_n \in (N \cup T)^*$ en $\alpha_1 \underset{G}{\Rightarrow} \alpha_2, \alpha_2 \underset{G}{\Rightarrow} \alpha_3, \ldots, \alpha_{n-1} \underset{G}{\Rightarrow} \alpha_n$ dan schrijven we $\alpha_1 \underset{G}{\Rightarrow} \alpha_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \alpha_n$ of korter $\alpha_1 \overset{+}{\underset{G}{\Rightarrow}} \alpha_n$ [lees: α_1 gene-

reert α_n in één of meer stappen]. Dus is $\overset{+}{\underset{G}{\Rightarrow}}$ de transitieve insluiting van de relatie $\underset{G}{\Rightarrow}$. De reflexieve insluiting van $\overset{+}{\underset{G}{\Rightarrow}}$ wordt weergegeven door $\overset{*}{\underset{G}{\Rightarrow}}$ en dus geldt $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_n$ desd als $\alpha_1 = \alpha_n$ of $\alpha_1 \overset{+}{\underset{G}{\Rightarrow}} \alpha_n$.

Als $\alpha \in (N \cup T)^*$ zodanig is dat $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ dan heet α een *sequentiële vorm* van G. Een *zin* uit G is iedere sequentiële vorm in T^*, dat wil zeggen een terminale string die uit S kan worden afgeleid. De *taal gegenereerd door* G, $L(G)$ is gedefinieerd als de verzameling van alle zinnen uit G. Dus geldt

$$L(G) = \{x \in T^* | S \overset{*}{\underset{G}{\Rightarrow}} x\}.$$

In veel gevallen is de grammatica G waarnaar we verwijzen duidelijk uit de context. In dergelijke gevallen kunnen we het subscript G weglaten uit $\underset{G}{\Rightarrow}$, $\overset{+}{\underset{G}{\Rightarrow}}$, $\overset{*}{\underset{G}{\Rightarrow}}$ en dus schrijven $\Rightarrow$, $\overset{+}{\Rightarrow}$, $\overset{*}{\Rightarrow}$.

Als we de voorbeelden G_1 en G_2 die we hierboven behandelden, gebruiken dan kan de lezer verifiëren dat

$$L(G_1) = \{a^n | n \geqslant 1\} \cup \{b^n | n \geqslant 1\}$$

en

$$L(G_2) = \{a^n b^n c^n | n \geqslant 1\}$$

Het kan voorkomen dat twee verschillende grammatica's G en G' dezelfde taal $L(G) = L(G')$ genereren. In dit geval heten de grammatica's *equivalent*. We geven een voorbeeld van een grammatica G_3 die equivalent is met G_1

$$S \to aA \,|\, bB \,|\, a \,|\, b$$
$$A \to aA \,|\, a$$
$$B \to bB \,|\, b$$

Frasestructuur grammatica's vormen een algemeen toepasbaar mechanisme voor het genereren van talen en Turingmachines zijn geschikt voor het herkennen daarvan. Het volgende resultaat komt daarom waarschijnlijk niet als een grote verrassing, maar het is wel een verdere onderbouwing van de *these van Church*.

Stelling 4.9

$L \subseteq T^*$ wordt gegenereerd door een PSG, $G = (N,T,P,S)$ desd als L r.e. is.

Bewijs. $\Rightarrow$: We beschrijven een nondeterministische TM, M zodanig dat $L = L(M)$. Het tape alfabet van M bevat $N \cup T$ en ook het symbool $\wedge$. Van dit laatste symbool veronderstellen we dat het geen element van $N \cup T$ is en verder introduceren we een speciaal symbool $\$ \notin N \cup T \cup \{\wedge\}$. Initieel staat op de tape van M een string $x \in T^*$. M kopieert deze naar de locaties $-|x|,\ldots,-1$ en merkt locatie 0 met het symbool \$ en schrijft vervolgens het symbool S op locatie 1. M simuleert nu nondeterministisch een afleiding binnen G op het positieve gedeelte van haar tape. Veronderstel dat op een bepaald moment de string $A_1A_2\ldots A_k$, $A_i \in N \cup T$, $i = 1,\ldots,k$ staat op de locaties $1,2,\ldots,k,k+1$. M kiest nu op nondeterministische wijze de locaties $i > 0$ en $j \geqslant i$. Mocht i of j het symbool $\wedge$ bevatten dan wordt er opnieuw een keuze gemaakt. M onderzoekt nu de substring $A_iA_{i+1}\ldots A_j$ en als dit het linkerlid is van een produktieregel in P dan wordt deze vervangen door het bijbehorende rechterlid. Het is daarbij mogelijk dat de string $A_{j+1}\ldots A_k$ naar links of naar rechts moet worden verschoven om ruimte te maken of om lege ruimte op te vullen. Het proces wordt herhaald totdat de simulatie van de afleiding kan stoppen omdat de tape geen nonterminale symbolen meer bevat. De tape bevat nu de string $x\$y$ voor de een of andere $x,y \in T^*$. De TM stopt nu als $x = y$, maar als dat niet het geval is dat wordt de string y verwijderd, vervangen door S en de simulatie wordt opnieuw begonnen.

$\Leftarrow$: We mogen veronderstellen dat L geaccepteerd wordt door een eenzijdig begrensde TM, $M = (Q,\Sigma,T,P,q\ ,F)$. We mogen verder aannemen (zie opgave 4.4) dat M nooit een leeg symbool op haar tape zet. We zullen nu een PSG, G construeren die L genereert. Het komt er op neer dat G twee kopieën van een string $x \in T^*$ genereert en de acties van M op één van de twee strings uitvoert. Als M de string accepteert dan converteert G de eerste kopie van de string naar een rij terminale symbolen. Als M de string niet accepteert dan wordt geen terminale string geproduceerd.

Formeel definiëren we de niet-terminale symbolen van G als

$([T \cup \{\varepsilon\}] \times \Sigma) \cup Q \cup \{S_1, S_2, S_3\}$ waarbij $S_1, S_2, S_3 \notin Q$ terwijl S_1 het startsymbool voorstelt. De produktieregels luiden nu als volgt.

Initialisatie produktieregels

$S_1 \to q_0 S_2$
$S_2 \to [a,a]S_2$ voor elke $a \in T$
$S_2 \to S_3$
$S_3 \to [\varepsilon, \wedge]S_3$
$S_3 \to \varepsilon$

Simulatie produktieregels

$[b, Z]q[a,X] \to q'[b, Z][a, Y]$ voor alle $a,b \in T \cup \{\varepsilon\}, Z \in \Sigma$ en voor alle $q, q' \in Q, X, Y \in \Sigma$ zodat $P(q, X) = (q', Y, L)$

$q[a,X] \to [a,Y]q'$ voor alle $a \in T \cup \{\varepsilon\}$ en voor alle $q, q' \in Q, X, Y \in \Sigma$ zodat $P(q,X) = (q', Y, R)$

$q[a,X] \to q'[a,Y]$ voor alle $a \in T \cup \{\varepsilon\}$ en voor alle $q, q' \in Q, X, Y \in \Sigma$ zodat $P(q,X) = (q', Y, 0)$.

Finale produktieregels

$[a,X]q \to qaq$
$q[a,X] \to qaq$
$q \to \varepsilon$
en voor alle $a \in T \cup \{\varepsilon\}, X \in \Sigma$ en $q \in F$

Nu geldt $S_1 \overset{*}{\Rightarrow} q_0[a_1,a_1][a_2,a_2]\ldots[a_n,a_n]S_2$ voor elke $a_i \in T$, $i = 1,2,\ldots,n$. Veronderstel dat M $a_1a_2\ldots a_n \in T^*$ accepteert dan is er een $k \geqslant n$ zodat M alleen de locaties $1,2,\ldots,k$ gebruikt. Uit S_1 kunnen we de string $q[a_1,a_1][a_2,a_2]\ldots[a_n,a_n][\varepsilon,\wedge]^m$ genereren met $m = k-n$ door alleen van de initialisatieregels gebruik te maken. Vervolgens simuleren we, gebruik makend van de simulatie produktieregels de actie van M op de tweede componenten van de geordende paren, waaruit de string bestaat. Door middel van inductie naar het aantal stappen in de afleiding kunnen we nu aantonen dat $q_0[a_1,a_1][a_2,a_2]\ldots[a_n,a_n][\varepsilon,\wedge]^m \overset{*}{\Rightarrow} [a_1,X_1][a_2,X_2]\ldots[a_{r-1},X_{r-1}]q[a_r,X_r]\ldots[a_{n+m},X_{n+m}]$ desd als de TM, M gegeven de input $a_1a_2\ldots a_n$ een configuratie $(q,r,X_1X_2\ldots X_{r-1},X_r,front(X_{r+1}\ldots X_{n+m}))$ kan bereiken. Alleen wanneer de TM vervolgens een eindtoestand $q \in F$ bereikt, kunnen de

finale produktieregels uit de grammatica worden gebruikt om $a_1a_2...a_n$ te genereren. Op deze wijze bewijzen we dat $S_1 \overset{*}{\Rightarrow} a_1a_2...a_n$ desd als $a_1a_2...a_n \in T(M)$. En dus genereert M de taal $L = T(M)$.

Twee belangrijke beslissingsproblemen staan in direct verband met PSG's.

Elementprobleem voor PSG's (Membership problem MPSG)
Gegeven: Een PSG, $G = (N, T, P, S)$ en een string $x \in T^*$
Gevraagd: Is $x \in L(G)$?

en het

Leegheidsprobleem voor PSG's (Emptiness problem EPSG)
Gegeven: Een PSG, $G = (N,T,P,S)$
Gevraagd: Is $L(G) = \phi$?

Men kan eenvoudig aantonen dat een gevolg van stelling 4.9 is dat deze twee problemen allebei onoplosbaar zijn. MPSG is in principe gelijk aan het halting probleem. EPSG is onoplosbaar omdat men als volgt kan aantonen dat MPSG$\sim$EPSG. Gegeven $G = (N,T,P,S)$ en een string $x \in T^*$ kunnen we een G' construeren die $L(G) \cap \{x\}$ genereert door gebruik te maken van de stellingen 4.3(2) en 4.9. Nu geldt $x \in L(G)$ desd als $L(G') \neq \phi$. In oefening 4.5 behandelen we enkele andere onoplosbare beslissingsproblemen die te maken hebben met PSG's.

CONTEXT-GEVOELIGE GRAMMATICA'S

PSG's werden voor het eerst ingevoerd door Noam Chomsky in zijn klassieke artikel uit 1956, (Chomsky, 1956). Eigenschappen werden bewezen in een tweede artikel, (Chomsky, 1959). Deze twee artikelen bereidden de weg voor voor een hele theorie over formele talen. De algemene PSG's die we eerder definieerden, worden meestal *type 0 grammatica's* genoemd. Dit is het meest algemene type en het genereert de meest algemene klasse van talen, namelijk de r.e. talen. In deze paragraaf en in de volgende twee behandelen we restricties die we op de grammatica

kunnen toepassen en laten we zien welke invloed deze restricties hebben op de eigenschappen van de gegenereerde talen. Op deze wijze leiden we een hiërarchie van talen af, die de *Chomsky hiërarchie* wordt genoemd. Deze hiërarchie wordt uitgebreid behandeld in het klassieke werk over formele talen *Formal languages and their relation to automata* (Hopcroft & Ullman, 1969).

Leggen we aan elke produktie $\alpha \to \beta$ van een PSG, $G = (N,T,P,S)$ een restrictie op zo dat $|\alpha| \leqslant |\beta|$ dan heet de grammatica een *context-gevoelige grammatica* (Context-sensitive Grammar of CSG). De taal die door deze grammatica wordt gegenereerd heet dan ook een *context-gevoelige taal* of *een type 1 taal*. Gevolg van de definitie is natuurlijk dat de lege string ε niet in een context-gevoelige taal kan voorkomen omdat elke string die kan worden afgeleid uit het startsymbool S minimaal lengte 1 moet hebben. Meestal laat men echter toch ε toe in een context-gevoelige taal door de definitie van CSG's uit te breiden met de produktieregel $S \to \varepsilon$, onder de voorwaarde dat S niet voorkomt als een substring in enig rechterlid van een produktieregel. Wij zullen dit hier ook doen. Als nu L een CSL is die werd gegenereerd door de grammatica $G = (N, T, P, S)$, terwijl $\varepsilon \notin L$ dan kunnen we gemakkelijk de CSG

$$G' = (N \cup \{S'\}, T, P \cup \{S' \to S. S' \to \varepsilon\}, S'), S' \notin N \cup T$$

construeren die $L \cup \{\varepsilon\}$ genereert.

Het volgende resultaat is van belang omdat eruit blijkt dat het elementprobleem (MCSG) voor CSG's oplosbaar is.

Stelling 4.10

Als $G = (N,T,P,S)$ een CSG is dan is $L(G)$ een recursieve taal.

Bewijs. Door inspectie van G kan men vaststellen of $\varepsilon \in L(G)$. Beschouw nu $x \in T^+$ zodanig dat $|x| = n$. We hebben nu een algoritme nodig dat kan controleren of $x \in L(G)$. Zij $T^n_m = \{\alpha | \alpha \in (N \cup T)^+,$ $|\alpha| \leqslant n$ en $S \overset{*}{\Rightarrow} \alpha$ door middel van een afleiding van hoogstens m stappen$\}$. Dan is $T^n_0 = S$ en kunnen we T^n_m uit T^n_{m-1} berekenen door gebruik te maken van $T^n_m = T^n_{m-1} \cup \{\alpha|$ voor een $\beta \in T^n_{m-1}$, $\beta \Rightarrow \alpha$ en $|\alpha| \leqslant n\}$. Als $S \overset{*}{\Rightarrow} \alpha$ en $|\alpha| < n$ dan zit α in T^n_m voor de een of andere

m; als uit S niet α wordt afgeleid of als $|\alpha| > n$ dan zit α niet in T^n_m voor enige m.

De TM die vaststelt of $x \in L(G)$ construeert $T^n_0, T^n_1, T^n_2, \ldots$ met $n = |x|$. Uit de definitie volgt dat $T^n_i \subseteq T^n_{i+1}$. Ook geldt dat als $T^n_i = T^n_{i+1}$ dan is $T^n_{i+1} = T^n_{i+2} = \ldots$. We construeren zo dus een *ketting* $T^n_0 \subset T^n_1 \subset T^n_2 \ldots \subset T^n_i = T^n_{i+1} = \ldots$. Nu geldt dat $\#(T^n_i) \leqslant k^n$ met $k = \#(N \cup T)$ en dus moet de ketting ooit een i bereiken met $T^n_i = T^n_{i+1}$. Het algoritme dat controleert of $x \in L(G)$ kan dus stoppen zodra het dit stadium bereikt. Dan geldt $x \in L(G)$ desd als $x \in T^n_i$.

Helaas is de omkering van deze stelling niet waar. Dat wil zeggen er bestaan recursieve talen die niet context-gevoelig zijn. Het bewijs hiervan is niet moeilijk. In oefening 4.7 wordt de lezer gevraagd aan te tonen dat er een aftelbaar oneindige hoeveelheid frasestructuur grammatica's bestaat met $N \subset \{A_i | i \geqslant 0\}$, $S = A_0$ en $T = \{0,1\}$. Verder kan ook bewezen worden dat de verzameling CSG's binnen deze verzameling grammatica's eveneens aftelbaar oneindig is en dus kan worden opgesomd als $G_1, G_2, G_3, \ldots$. Eerder toonden we aan dat we de strings in $\{0,1\}^*$ kunnen enumereren als $x_1, x_2, x_3, \ldots$, Op grond hiervan kunnen we de taal $L = \{x_i \; x_i \notin L(G_i)\}$ definiëren. L is zeker recursief omdat men gegeven een $x \in \{0,1\}^*$ kan bepalen voor welke i geldt dat $x_i = x$. Vervolgens kan men G_i genereren en dan met behulp van stelling 4.10 testen of $x_i \in L(G_i)$. L is echter geen context-gevoelige taal. Als dit wel het geval was dan gold $L = L(G_j)$ voor zekere j. Beschouw nu x_j. Als $x_j \in L$ dan geldt $x_j \in L(G_j)$ en dus volgt uit de definitie van L dat $x_j \notin L$. Als $x_j \notin L$ dan is $x_j \notin L(G_j)$ en dus geldt opnieuw wegens de definitie van L dat $x_j \in L$. In beide gevallen krijgen we een tegenspraak en we kunnen dus de volgende stelling formuleren.

Stelling 4.11

Er bestaan recursieve talen die niet context-gevoelig zijn.

Omdat in een CSG, G elke produktieregel $\alpha \to \beta$ (met uitzondering van het speciale geval $S \to \varepsilon$) voldoet aan $|\alpha| \leqslant |\beta|$, is het duidelijk dat als $\gamma_1\alpha\gamma_2 \Rightarrow \gamma_1\beta\gamma_2$ dat dan $|\gamma_1\alpha\gamma_2| \leqslant |\gamma_1\beta\gamma_2|$. Tijdens de afleiding van een string $x \in T^+$ heeft elke zinsvorm dus een lengte $\geqslant 1$ en $\leqslant |x|$. Beschouw nu een nondeterministische TM zoals geconstrueerd werd in het bewijs van het eerste deel van stelling 4.9. Als G een CSG is dan zal de TM wegens de restrictie op de lengte van de zinsvorm die werd gebruikt om x af te leiden, iedere $x \in L(G)$ accepteren en daarbij alleen gebruik maken van de locaties $-|x|-1, -|x|, -|x|+1, \ldots, |x|, |x|+1$. De locaties $-|x|-1$ en $|x|+1$ bevatten steeds een speciaal symbool dat dient om het einde van de string aan te geven. Als het niet mogelijk is een zinsvorm te plaatsen in de locaties $1, 2, \ldots, |x|$, dan weten we dat deze zinsvorm niet kan worden gebruikt voor het afleiden van x. Zodra we proberen van locatie $|x|+1$ gebruik te maken om er een symbool uit een zinsvorm in op te slaan, weten we dat we deze afleiding kunnen opgeven. Het positieve deel van de tape kan worden schoongemaakt en we kunnen de simulatie opnieuw beginnen met het startsymbool S in locatie 1.

Een nondeterministische Turingmachine die voor alle inputs x alleen gebruik maakt van $k_1|x|+k_2$ locaties op de tape, waarbij k_1 en k_2 twee vaste gehele getallen zijn, heet een *lineair begrensde automaat* (LBA). We lieten zien dat elke context-gevoelige taal L wordt geaccepteerd door een lineair begrensde automaat, want de TM die we beschreven gebruikt nooit meer dan $2|x|+2$ locaties voor het accepteren van $x \in L$. Het omgekeerde resultaat geldt ook: elke taal die wordt geaccepteerd door een LBA is noodzakelijkerwijs context-gevoelig. Kuroda (1964) geeft een effectieve constructie waarmee dit wordt bewezen. Gevolg hiervan is dat we ook de volgende stelling kunnen bewijzen.

Stelling 4.12

Het leegheidsprobleem voor context-gevoelige grammatica's (Emptiness problem for Context-sensitive Grammars of ECSG) is onoplosbaar.

Bewijs. We tonen aan dat PCP $\sim$ ECSG. Stel $\underset{\sim}{x} = (x_1, x_2, \ldots, x_n)$ en $\underset{\sim}{y} = (y_1, y_1, \ldots, y_n)$ zijn twee n-tupels van strings in T^+ die een instan-

tie I van PCP definiëren. We kunnen nu een LBA M construeren die, gegeven een inputstring $z \in T^+$, op nondeterministische wijze een rij gehele getallen $i_1, i_2, \ldots, i_m$ $(1 \leqslant m \leqslant |z|$ en $1 \leqslant i_j \leqslant n$ voor alle $j)$ genereert. M gaat vervolgens na of $z = x_{i_1} x_{i_2} \ldots x_{i_m} = y_{i_1} y_{i_2} \ldots y_{i_m}$. Als dit het geval blijkt dan accepteert M z. Er geldt dus dat z door deze LBA wordt geaccepteerd dan en slechts dan als z een oplossing voor I voorstelt. We weten verder dat er een CSG, G moet bestaan, zodanig dat $L(G)$ de taal is die door M wordt geaccepteerd. Dus geldt dat $L(G) \neq \emptyset$ desd als I een oplossing heeft. Omdat er een effectieve constructie bestaat voor M en G is hiermee aangetoond dat PCP$\sim$ECSG, hetgeen te bewijzen was.

CONTEXTVRIJE GRAMMATICA'S

Een *contextvrije* of *type-2* grammatica (Context-free Grammar of CFG) is een frasestructuur grammatica $G = (N, T, P, S)$ waarin iedere produktie P van de vorm $A \to \beta$, $A \in N$, $\beta \in (N \cup T)^*$ is. Een taal die door zo'n grammatica wordt gegenereerd heet een contextvrije taal of een type-2 taal, (CFL). De volgende grammatica G_4 is een voorbeeld van een contextvrije grammatica:

$$S \to AB \mid aC$$
$$A \to aA \mid b \mid BB$$
$$B \to bSS \mid aB \mid \varepsilon$$
$$C \to ab$$

Deze grammatica bevat de produktie $B \to \varepsilon$ en is daarom dus niet contextgevoelig. We kunnen echter een beperking aanbrengen ten aanzien van *ε-produkties*. Als een grammatica is het geheel geen ε-produkties bevat dan heet deze *ε-vrij*. Als een CFG, G ε-produkties bevat dan is het mogelijk een ε-vrije CFG, G' te construeren zodanig dat $L(G') = L(G) \setminus \{\varepsilon\}$. De constructie is niet moeilijk: als $G = (N, T, P, S)$ dan is $G' = (N, T, P', S)$. P' wordt volgens de volgende regels uit P geconstrueerd.

(1) Plaats alle ε-vrije produkties uit P in P'.

(2) Bepaal alle niet-terminale grootheden $A \in N$, waarbij $A \overset{*}{\underset{G}{\Rightarrow}} \varepsilon$.

Dergelijke niet-terminale grootheden worden ε-*genererend* genoemd. Voeg nu voor elke produktie $p \in P$ waarin een of meer ε-genererende niet-terminale grootheden als substrings in het rechterlid voorkomen, aan P' alle vrije produkties toe die uit p kunnen worden geconstrueerd door een of meer van deze ε-genererende niet-terminale grootheden weg te laten.

In ons voorbeeld zijn S, A en B ε-genererend, maar C is dat niet. Als we de bovenstaande regels toepassen dan krijgen we de volgende contextvrije grammatica

$$S \to AB \mid aC \mid A \mid B$$
$$A \to aA \mid b \mid BB \mid a \mid B$$
$$B \to bSS \mid ab \mid b \mid bS \mid a$$
$$C \to ab$$

Deze CFG genereert $L(G_4)\setminus\{\varepsilon\}$.

Bovenstaande constructie brengt altijd uit een CFG, $G = (N,T,P,S)$ een ε-vrije CFG, $G' = (N,T,P',S)$ voort met $L(G)\setminus\{\varepsilon\}$. Het bewijs is niet moeilijk en berust op het feit dat voor alle $A \in N$ en $x \in T^+$ geldt dat $A \overset{*}{\underset{G}{\Rightarrow}} x$ desd als $A \overset{*}{\underset{G'}{\Rightarrow}} x$. Als $\varepsilon \in L(G)$ dan is G' niet equivalent met G, maar we kunnen wel ε opnieuw invoeren en verder kunnen we ervoor zorgen, door een nieuwe niet-terminale grootheid S' in te voeren, dat het startsymbool nooit voorkomt als een substring in een rechterlid van een produktieregel. Definieer daartoe $G'' = (N \cup \{S'\},T,P'',S')$ waarbij $P'' = P \cup \{S' \to \varepsilon, S' \to S\}$. In ons voorbeeld geldt dat $S \underset{G_4}{\Rightarrow} AB \underset{G_4}{\Rightarrow} BBB \overset{*}{\underset{G_4}{\Rightarrow}} \varepsilon$ en dus $\varepsilon \in L(G_4)$. Hiermee hebben we G_4 geconstrueerd met de produktieregels

$$S' \to \varepsilon \mid S$$
$$A \to AB \mid aC \mid A \mid B$$
$$A \to aA \mid b \mid BB \mid a \mid B$$
$$B \to bSS \mid aB \mid b \mid bS \mid a$$
$$C \to ab$$

met S' als nieuw startsymbool.

Voor elke willekeurige CFG, G bestaat er een algoritme om vast te stellen of $\varepsilon \in L(G)$, (zie oefening 4.13). Als $\varepsilon \notin L(G)$ dan is de hierboven geconstrueerde G' equivalent met G en als $\varepsilon \in L(G)$ dan is G''

equivalent met G. In beide gevallen is zo geconstrueerde grammatica in overeenstemming met onze definitie van een context-gevoelige grammatica. Hiermee hebben we de volgende stelling bewezen.

Stelling 4.13

Elke context-vrije taal is een context-gevoelige taal.

De omgekeerde bewering is niet waar. Bijvoorbeeld $\{a^n b^n c^n \mid n \geqslant 1\}$ is context-gevoelig, zoals we al eerder aantoonden, maar is niet context-vrij. Het bewijs hiervan wordt gegeven in onze *Inleiding in de Theorie van de Formele Talen* (Rayward-Smith, 1983). Daarin tonen we ook aan dat de acceptoren voor CFL's non-deterministische stack-automaten (Nondeterministic Pushdown Automata of NDPA's) zijn. Dit zijn beperkte non-deterministische Turingmachines met eenzijdig onbegrensde tapes. De input wordt geschreven op de eerste tape en de kop van deze tape kan alleen lezen en kan niet naar links bewegen. De machine mag niet in een eindtoestand geraken voordat de uiterst linkse lege positie op deze tape is gelezen. De tweede tape wordt als *stack* gebruikt, dat wil zeggen als positie k de uiterst linkse lege positie op de tape is dan zijn ook alle posities $>k$ leeg. De kop kan locatie k-1 lezen, de inhoud door het lege symbool vervangen en naar links bewegen, (dit is de *pop*-operatie). De kop kan ook locatie k lezen en daar een ander symbool schrijven, (dit is de *push*-operatie). Elke taal die door een NDPA geaccepteerd wordt is noodzakelijk context-vrij en omgekeerd bestaat er voor elke context-vrije taal een NDPA die deze taal accepteert. Maken we de machine deterministisch dan is dit niet langer meer het geval: er bestaan context-vrije talen die door geen enkele deterministische stack-automaat worden geaccepteerd.

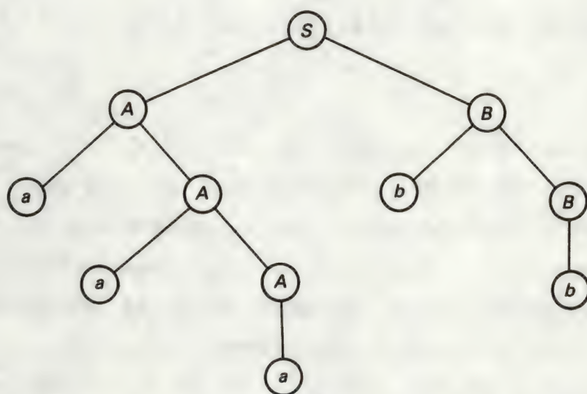
Omdat we uit elke CFG een CSG kunnen construeren en omdat het elementprobleem voor CSG's oplosbaar is, is ook het elementprobleem voor CFG's oplosbaar. Een efficiënte oplossing voor dit probleem is voor de informatica van groot belang. De *syntaxis* (de grammaticale structuur) van programmeertalen wordt meestal beschreven in Backus-Naur Form (BNF) of in syntaxisdiagrammen. Beide notaties zijn equivalent

met een context-vrije grammatica. Om te controleren of een gegeven programma dat is geschreven in de programmeertaal ook geconstrueerd is overeenkomstig de bijbehorende syntaxis moet het elementprobleem worden opgelost. In de praktijk worden meestal restricties aan de grammatica toegevoegd om een efficiënte oplossing van dit probleem mogelijk te maken.

Zij $G = (N, T, P, S)$ een willekeurige CFG en veronderstel dat $L(G) \neq \emptyset$. Op grond van het voorgaande mogen we veronderstellen dat G context-gevoelig is. Nu bestaat er bij elke string $x \in T^+$ die door G wordt gegenereerd een afleidingsboom. We definiëren de diepte van een dergelijke boom als de lengte van het langste pad vanuit de wortel naar een blad. Voor G_5 met de produktieregels

$$S \rightarrow AB$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow bB \mid b$$

is in figuur 4.2 voor $a^3b^2 \in L(G_5)$ de afleidingsboom met diepte vier weergegeven.



Figuur 4.2

Als een afleidingsboom die werd geconstrueerd door middel van een CFG, $G = (N, T, P, S)$ diepte n heeft dan moet er een pad met knooppunten $A_1, A_2, \ldots, A_n, a$ bestaan met $S = A_1, A_2, \ldots, A_n \in N$ en $a \in T$. Als daarbij $n > \#(N)$ dan moeten twee of meer knooppunten op dit pad

gelijk zijn, bijvoorbeeld $A_i = A_j$ met $i < j$. We kunnen dan een nieuwe afleidingsboom construeren door de subboom met wortel A_i te vervangen door die met wortel A_j. Door herhaald toepassen van deze redenering kunnen we concluderen dat er een afleidingsboom voor een of andere $x \in L(G)$ met diepte $\leqslant \#(N)$ moet bestaan als er een afleidingsboom voor $x' \in L(G)$ met diepte $> \#(N)$ bestaat. Hieruit volgt

Stelling 4.14

Het leegheidsprobleem voor contextvrije grammatica's is oplosbaar. Dat wil zeggen bij elke CFG, $G = (N, T, P, S)$ bestaat er een algoritme dat vaststelt of $L(G) = \phi$.

Bewijs: Veronderstel dat de CFG, G context-gevoelig is. Het algoritme test dan eerst of $S \rightarrow \varepsilon$ een produktieregel in P is. Als dat het geval is dan geldt $L(G) \neq \phi$ omdat immers $\varepsilon \in L(G)$. Als $S \rightarrow \varepsilon$ geen produktieregel in P is dan construeren we alle mogelijke afleidingsbomen *maar alleen tot diepte* $\#(N)$. Daarvan bestaat een eindige hoeveelheid en geen van deze afleidingsbomen komt overeen met een afleiding van een terminale string dan en slechts dan als $L(G) = \phi$.

Een produktie $A \rightarrow \alpha$ in een CFG, $G = (N, T, P, S)$ heet *relevant* dan en slechts dan als er een afleiding voor een $x \in L(G)$ bestaat die van deze produktie gebruik maakt, dat wil zeggen dan en slechts dan als $S \overset{*}{\Rightarrow} \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \alpha \alpha_2 \overset{*}{\Rightarrow} x$ voor een $x \in L(G)$. Een niet-relevante produktie in G heet *irrelevant* en die willen we graag verwijderen uit G. Nu kunnen we voor elke $A \in N$ een nieuwe grammatica $G_A = (N, T, P, A)$ construeren. Als $L(G_A) = \phi$ dan is het onmogelijk een terminale string uit A te genereren door van produktieregels uit P gebruik te maken. We kunnen dus uit geen enkele zinsvorm uit G die A bevat een terminale string afleiden. $L(G)$ blijft dus ongewijzigd als we uit G alle produktieregels verwijderen met A in het linkerlid of met A in het rechterlid. Wel kunnen we dan nog irrelevante produkties overblijven in de grammatica omdat er $A \in N$ kunnen bestaan die terminale strings kunnen genereren maar die nooit in een zinsvorm kunnen

voorkomen. We kunnen een overeenkomstige redenering gebruiken als in het bewijs van stelling 4.14 om aan te tonen dat als A in een zinsvorm kan voorkomen er een zinsvorm moet bestaan die A bevat en die kan worden afgeleid met behulp van een partiële afleidingsboom met diepte $< \#(N)$. (De boom heet *partieel* omdat de eindpunten geen terminalen zijn.) Door al dit soort bomen te genereren kunnen we controleren of er soms een $A \in N$ is die niet in een zinsvorm kan voorkomen. Als dat zo is dan kan elke produktie waarin A voorkomt als irrelevant worden verwijderd.

Hoewel we irrelevante produkties uit een contextvrije grammatica kunnen verwijderen en het elementprobleem en het leegheidsprobleem kunnen oplossen, bestaan er toch een aantal niet oplosbare problemen ten aanzien van deze grammatica's. Bijvoorbeeld

Het lege doorsnede probleem voor contextvrije grammatica's. (EICFG of Empty Intersection Problem for Context-Free Grammars)
Gegeven: Twee CFG's, G_1 en G_2.
Gevraagd: Is $L(G_1) \cap L(G_2) = \emptyset$?

Stelling 4.15

Het lege doorsnede probleem voor CFG's is onoplosbaar.

Bewijs: We zullen aantonen PCP $\sim$ EICFG. Als een instantie I van PCP gegeven is door $\underset{\sim}{x} = (x_1, x_2, \ldots, x_n)$ en $\underset{\sim}{y} = (y_1, y_2, \ldots, y_n)$ twee n-tupels van strings in T^+ dan construeren we als volgt een bijbehorend voorkomen van EICFG. Zij $\{s_1, s_2, \ldots, s_n\}$ een rij van n verschillende symbolen $\notin T$ en definieer T' als $T' = T \cup \{s_1, s_2, \ldots, s_n\}$. Veronderstel verder dat $G_x = (\{S_x\}, T', P_x, S_x)$ en $G_y = (\{S_y\}, T', P_y, S_y)$ twee CFG's zijn zodanig dat P_x alle produkties $S_x \to x_i S_x s_i$ en $S_x \to x_i s_i$, $i = 1, 2, \ldots, n$ bevat, terwijl P_y alle produkties $S_y \to y_i S_y s_i$ en $S_y \to y_i s_i$, $i = 1, 2, \ldots, n$ bevat. Nu is $L(G_x) = \{x_{i_1} x_{i_2} \ldots x_{i_m} s_{i_m} \ldots s_{i_2} s_{i_1} | m \geqslant 1\}$ en evenzo is $L(G_x) = \{y_{i_1} y_{i_2} \ldots y_{i_m} s_{i_m} \ldots s_{i_2} s_{i_1} | m \geqslant 1\}$. Dus geldt $L(G_x) \cap L(G_y) \neq \emptyset$ desd als I een oplossing heeft. Omdat vanuit I G_x en G_y effectief te construeren zijn hebben we bewezen dat EICFG onoplosbaar is.

Er bestaan nog andere onoplosbare problemen ten aanzien van CFG's. Onder andere de volgende.

Totaliteitsprobleem voor CFG's (TCFG)
Gegeven: Een CFG, $G = (N,T,P,S)$.
Gevraagd: Is $L(G) = T^*$?

Equivalentieprobleem voor CFG's (ECFG)
Gegeven: Twee CFG's G_1 en G_2.
Gevraagd: Is $L(G_1) = L(G_1)$?

Eindige doorsnede probleem voor CFG's (FICFG)
Gegeven: Twee CFG's G_1 en G_2.
Gevraagd: Is $L(G_1) \cap L(G_2)$ eindig?

De bewijzen van deze en andere resultaten staan vermeld in Bar-Hillel, Perles & Shamir (1961) en in Ginsburg & Rose (1963).

REGULIERE GRAMMATICA'S

Een frasestructuur grammatica (PSG) $G = (N,T,P,S)$ heet een *reguliere grammatica* als:

(1) als er een ε-produktie in G bestaat dan is deze van de vorm $S \rightarrow \varepsilon$ en dan komt S niet voor als een substring in het rechterlid van enig andere produktieregel in P;

(2) alle andere produkties zijn ofwel van de vorm

$A \rightarrow a$ met $A \in N$ en $a \in T$

ofwel van de vorm

$A \rightarrow aB$ met $A,B \in N$, $a \in T$.

Een taal heet een *reguliere taal* (of ook wel een *reguliere verzameling*) dan en slechts dan als de taal gegenereerd wordt door een of andere reguliere grammatica. Duidelijk is dat L een reguliere taal is desd als $L \backslash \{\varepsilon\}$ gegenereerd wordt door een ε-vrije reguliere grammatica. Ook

is elke reguliere grammatica noodzakelijkerwijs context-vrij en dus is elke reguliere taal een context-vrije taal.

Een eenvoudig voorbeeld van een reguliere grammatica is de grammatica G_6 met als produktieregels

$$S \rightarrow aA \mid bB$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow bB \mid b$$

Er geldt $L(G_6) = \{a^n \mid n > 1\} \cup \{b^n \mid n > 1\}$.

Elke reguliere taal wordt geaccepteerd door een niet-deterministische Turingmachine met een eenzijdig onbegrensde tape waarlangs een leeskop alleen naar rechts beweegt. Een dergelijke machine heet een *nondeterministische eindige toestandsautomaat*, (Nondeterministic Finite State Automaton of NFSA).
In Rayward-Smith (1983) wordt de volgende stelling bewezen.

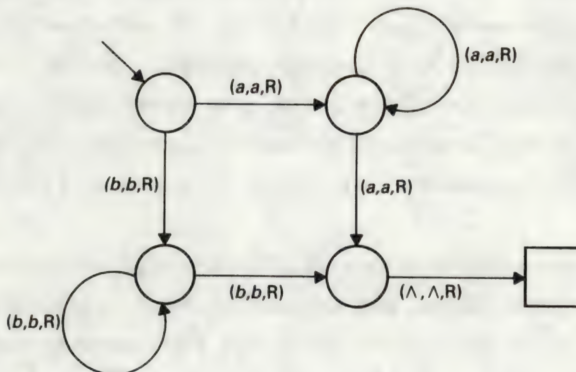
Stelling 4.16

De volgende definities zijn equivalent:

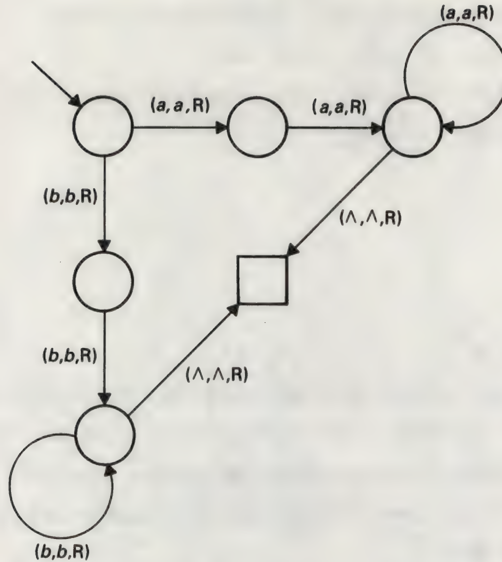
(i) $L \subseteq T^*$ wordt gegenereerd door een reguliere grammatica;

(ii) $L \subseteq T^*$ wordt geaccepteerd door een nondeterministische eindige automaat (NFSA);

(iii) $L \subseteq T^*$ wordt geaccepteerd door een deterministische eindige automaal (DFSA).



Figuur 4.3



Figuur 4.4

De NFSA weergegeven in figuur 4.3 is een voorbeeld van een automaat die $L(G_6)$ accepteert. Merk op dat in dit voorbeeld alle verbindingen gelabeld zijn met een tripel van de vorm (a, a, R) voor een of andere $a \in T \cup \{\wedge\}$. Dit is altijd het geval bij een eindige automaat omdat de kop alleen kan lezen en alleen naar rechts beweegt.

In figuur 4.4 is een DFSA weergegeven die equivalent is met de NFSA van figuur 4.3 en die ook $L(G_6)$ accepteert. Omdat reguliere grammatica's per definitie altijd context-vrij zijn is het elementprobleem, het leegheidsprobleem en zijn de eindigheidsproblemen voor reguliere grammatica's allen oplosbaar. Door middel van de constructies van geschikte eindige automaten kan men ook aantonen dat het totaliteitsprobleem, het equivalentieprobleem en de problemen van de lege doorsnede en van de eindige doorsnede eveneens oplosbaar zijn. De geïnteresseerde lezer wordt verwezen naar Hopcroft & Ullman (1969) voor de bewijzen.

De reguliere grammatica's maken de Chomsky hiërarchie van grammatica's volledig en staan daarom ook bekend als type 3 grammatica's. In figuur 4.5 geven we de totale hiërarchie met bijbehorende acceptoren en al dan niet oplosbare problemen weer.

	GRAMMATICA	frase structuur	context-gevoelig	context-vrij	regulier
	TAAL	recursief aftelbaar	context-gevoelig	context-vrij	
	ACCEPTOR	Turingmachine	Lineair begrensde automaat	nondeterministische stack automaat	eindige automaat
PROBLEEM-OPLOS-BAARHEID	ELEMENT	Onoplosbaar	Onoplosbaar	Oplosbaar	Oplosbaar
	LEEGHEID	Onoplosbaar	Onoplosbaar	Oplosbaar	Oplosbaar
	EINDIGHEID	Onoplosbaar	Onoplosbaar	Oplosbaar	Oplosbaar
	TOTALITEIT	Onoplosbaar	Onoplosbaar	Onoplosbaar	Oplosbaar
	EQUIVALENTIE	Onoplosbaar	Onoplosbaar	Onoplosbaar	Oplosbaar
	LEGE DOORSNEDE	Onoplosbaar	Onoplosbaar	Onoplosbaar	Oplosbaar
	EINDIGE DOORSNEDE	Onoplosbaar	Onoplosbaar	OnOplosbaar	Oplosbaar

Figuur 4.5 De Chomsky hiërarchie

OEFENINGEN

1. Toon aan dat voor elke eindige verzameling T de verzameling van TM's van de vorm (Q,Σ,T,P,q_0,F) met $Q\subset\{q_i|i\geqslant 0\}$ en $\Sigma\subset T\cup\{s_i|i\geqslant 0\}$, aftelbaar oneindig is. Laat zien dat elke recursief aftelbare taal herkend wordt door tenminste één van deze Turing-machines.

2. Toon aan dat $L = \{context(x,rev(x))|x\in\{0,1\}^*\}$ een recursieve verzameling is door een (deterministische) TM te construeren die χ_L berekent.

3. Bewijs stelling 4.7, dat wil zeggen dat L geaccepteerd wordt door een (deterministische) TM desd als L recursief aftelbaar is.

4. Toon aan dat als L wordt geaccepteerd door een deterministische eenzijdig onbegrensde TM dat dan L kan worden geaccepteerd door een dergelijke machine die nooit een leeg symbool op zijn tape schrijft. Hint: voeg aan het hulpalfabet een 'pseudo-leeg' symbool toe.

5. Toon aan dat de volgende problemen onoplosbaar zijn

   Eindigheidsprobleem voor frasestructuur grammatica's (PSG's)
   Gegeven: Een PSG, $G = (N,T,P,S)$.
   Gevraagd: Is $L(G)$ eindig?

Totaliteitsprobleem voor PSG's

Gegeven: Een PSG, $G = (N, T, P, S)$.

Gevraagd: Is $L(G) = T^*$?

Equivalentieprobleem voor PSG's

Gegeven: PSG's $G_1 = (N_1, T, P_1, S_2)$ en $G_2 = (N_2, T, P_2, S_2)$

Gevraagd: Is $L(G_1) = L(G_2)$?

6. Toon aan dat het *totaliteitsprobleem voor CSG's* onoplosbaar is.

7. Laat zien dat er aftelbaar oneindig veel frasestructuur grammatica's bestaan met $N \subset \{A_i | i \geqslant 0\}$, $S = A_0$ en $T = \{0,1\}$. Bewijs dat ook het aantal van deze grammatica's dat context-gevoelig is aftelbaar oneindig is.

8. Bewijs dat er bij elke recursief enumereerbare taal een procedure bestaat voor het aftellen van de strings uit die taal, (dit is de reden dat de taal recursief enumereerbaar wordt genoemd). [Hint: gebruik stelling 4.7.]

9. Laat zien dat het stellen van $k_1 = 1$ in de definitie van lineair begrensde automaten geen invloed heeft op het vermogen van dergelijke machines om alle context-gevoelige talen te herkennen. Wat voor extra randvoorwaarden zou men aan k_2 kunnen opleggen?

10. Toon aan dat het *Halting probleem voor lineair begrensde automaten* oplosbaar is.

11. Toon aan dat het *Eindigheidsprobleem voor context-vrije grammatica's* oplosbaar is.

12. Bewijs formeel dat de constructie van een ε-vrije CFG, G' uit een CFG, G, zoals in dit hoofdstuk beschreven, inderdaad voldoet aan $L(G') = L(G) \backslash \{\varepsilon\}$.

13. Bewijs dat het probleem oplosbaar is om voor een gegeven CFG, $G = (N, T, P, S)$ vast te stellen of een willekeurige $A \in N$ al dan niet ε-genererend is. Leid hieruit af dat er een algoritme bestaat dat vaststelt of al dan niet $\varepsilon \in L(G)$.

14. Toon aan dat als L_1, L_2 context-vrije talen zijn, dat dan ook $L_1 \cup L_2$ en $L_1 L_2$ dit zijn. Construeer een voorbeeld om te laten zien dat $L_1 \cap L_2$ niet noodzakelijk context-vrij is. [Hint: maak gebruik van het feit dat $\{a^n b^n c^n | n \geqslant 1\}$ niet context-vrij is.]

15. Laat zien dat $\{a^n b^n | n \geqslant 1\}$ geen reguliere taal is, maar wel context-vrij is.

16. Construeer een DFSA die $L = \{a,b\}\{aa,ba\}\{b\}^*$ accepteert. Construeer vervolgens hieruit, of op een andere wijze, een DFSA die $L^c = \{a,b\}^* \backslash L$ accepteert.

5

Recursieve Functies

Though deep yet clear, though gentle yet not dull;
Strong without rage, without o'erflowing full.

Diep maar helder, zachtmoedig maar met pit;
Sterk zonder woede, een beker waar niet te veel in zit.

Sir John Denham
Copper's Hill

HET DEFINIEREN VAN FUNCTIES

In het algemeen berekent een TM, $M = (Q,\Sigma,T,P,q\ ,F)$ een partiële functie f_M: $T^* \to \Sigma^*$. In het vorige hoofdstuk bestudeerden we eigenschappen van de deelverzameling van T^* waarop een dergelijke functie is gedefinieerd. In dit hoofdstuk zullen we de functies zelf bestuderen. We zullen de klasse van functies beschrijven die Turing-berekenbaar zijn (TM-berekenbaar) en we zullen de eigenschappen van deze functies onderzoeken.

Omdat men in de informatica gewend is met binaire codes te werken, zullen we veronderstellen dat zowel de argumenten van onze functies als de resulterende waarden in binaire vorm gecodeerd zijn. Als $B = \{0,1\}^*$ de verzameling van alle binaire strings voorstelt dan willen we die n-aire functies $B^n \to B$, $(n \geqslant 1)$ onderzoeken die Turing-berekenbaar zijn. Uit hoofdstuk 2 weten we al dat dit niet alle functies op B betreft, dat wil zeggen: er bestaan functies $B^n \to B$ die niet Turing-berekenbaar zijn. Er bestaan zelfs niet aftelbaar oneindig veel functies op B die niet berekenbaar zijn (oefening 5.1).

We beginnen onze behandeling van berekenbare functies met het onderzoeken van manieren waarop we nieuwe berekenbare functies kunnen construeren uit functies waarvan we al weten dat zij berekenbaar zijn.

Beschouw allereerst eens de m-aire functie $h:B^m \to B$ en de m n-aire functies $g_1, g_2, \dots, g_m : B^n \to B$. Hieruit kunnen we een nieuwe functie f construeren, de *compositie* van $h, g_1, g_2, \dots, g_m$ en dit is eveneens een n-aire functie $B^n \to B$ gedefinieerd door

$$f(x_1, x_2, \dots, x_n) = h(g_1(x_1, x_2, \dots, x_n), g_2(x_1, x_2, \dots, x_n), \dots, g_m(x_1, x_2, \dots, x_n))$$

Als voor enig n-tupel $(x_1, x_2, \dots, x_n)$ een waarde van $g_i(x_1, x_2, \dots, x_n)$ ongedefinieerd is $(1 \leqslant i \leqslant m)$, of als h ongedefinieerd is op de argumenten

$$g_1(x_1, x_2, \dots, x_n), g_2(x_1, x_2, \dots, x_n), \dots, g_m(x_1, x_2, \dots, x_n)$$

dan is oof f voor dat n-tupel ongedefinieerd.

Stelling 5.1

Als $h:B^m \to B$ en $g_1, g_2, \dots, g_m : B^n \to B$ TM-berekenbare functies zijn dan is de compositie van $h, g_1, g_2, \dots, g_m$ dat ook.

Bewijs. Veronderstel dat de TM, M de functie h berekent en dat de TM's $M_1, M_2, \dots, M_m$, $g_1, g_2, \dots, g_m$ berekenen. Zonder verlies van algemeenheid mogen we veronderstellen dat de machines $M_1, M_2, \dots, M_m$ als input een n-tupel van elementen van B accepteren met gebruik van dezelfde codering $e((x_1, x_2, \dots, x_n))$. We construeren nu een TM, M' met $m+1$ tapes die de compositie van $h, g_1, g_2, \dots, g_m$ berekent. M' accepteert als input de codering $e((x_1, x_2, \dots, x_n))$ en kopieert die naar de tapes $2, 3, \dots, m+1$. M' simuleert vervolgens de werking van machine M_{i-1} op tape i, $2 \leqslant i \leqslant m+1$. Als al deze machines stoppen dan worden vervolgens hun outputs gekopieerd op tape 1 en gecodeerd in een geschikte vorm voor machine M. Nu simuleert M' de werking van M.

In figúur 2.4 lieten we zien dat $cons0:B \to B$ TM-berekenbaar is. Evenzo kan men aantonen dat $cons1:B \to B$ TM-berekenbaar is. Hierbij is

$cons1(x) = 1x.$

Volgens stelling 5.1 mogen we stellen dat functies zoals

$$cons00(x) = cons0(cons0(x))$$
$$cons01(x) = cons0(cons1(x))$$

eveneens TM-berekenbaar zijn.

De functie $nil{:}B \to B$, met

$$nil(x) = \varepsilon \quad \text{voor alle} \quad x \in B.$$

is zeker TM-berekenbaar. Vervolgens kunnen de functies *nul* en *een* worden gedefinieerd door van compositie gebruik te maken:

$$nul(x) = cons0(nil(x))$$
$$een(x) = cons1(nil(x))$$

en deze functies moeten dus ook TM-berekenbaar zijn.

Als we compositie gebruiken om aan te tonen dat een functie TM-berekenbaar is dan is het niet per se nodig dat elke g_i in de constructie n-air is. We spreken af dat we functies $g_1, g_2, \ldots, g_m$ mogen gebruiken die k-air zijn, $(1 \leqslant k \leqslant n)$. Dit wil zeggen dat niet noodzakelijk alle argumenten $x_1, x_2, \ldots, x_n$ voorkomen in de argumentenlijsten van alle functies $g_1, g_2, \ldots, g_m$. We zouden bijvoorbeeld kunnen hebben aangetoond dat de functies $concat{:}B^2 \to B$ en $omgek{:}B \to B$ beiden TM-berekenbaar zijn. We zouden dan kunnen concluderen dat de functie

$$f_1(x,y) = concat(omgek(x), omgek(y))$$

TM-berekenbaar is omdat f_1 uit *concat* en *omgek* via compositie is geconstrueerd. Dit is echter niet correct omdat volgens de definitie van compositie de binnenste functies in het rechterlid van een dergelijke vergelijking alle argumenten moeten bevatten.

Voor een correcte behandeling moeten we eerst vaststellen dat de binaire projectiefuncties δ_2^1 en δ_2^2, gedefinieerd door

$$\delta_2^1(x,y) = x$$
$$\delta_2^2(x,y) = y$$

beiden TM-berekenbaar zijn. (De constructie van een TM om deze functies te berekenen ligt voor de hand.) Dus als *omgek* TM-berekenbaar is dan geldt dat volgens stelling 5.1 ook voor

$$omgek1(x,y) = omgek(\delta_2^1(x,y)) = omgek(x)$$

en $$omgek2(x,y) = omgek(\delta_2^2(x,y)) = omgek(y)$$

Door opnieuw stelling 5.1 toe te passen leiden we af dat

$$\begin{aligned} f\ (x,y) &= concat(omgek1(x,y), omgek2(x,y)) \\ &= concat(omgek(x), omgek(y)) \end{aligned}$$

ook TM-berekenbaar is.

Algemener kunnen we gemakkelijk aantonen dat de *projectiefuncties* $\delta_n^i(B^n \to B(1 \leqslant i \leqslant n)$ gedefinieerd door

$$\delta_n^i(x_1,x_2,\ldots,x_n) = x_i$$

TM-berekenbaar zijn. Door deze functies te gebruiken bij de compositie met bekende functies kunnen we ervoor zorgen dat niet gebruikte argumenten komen te vervallen. We kunnen projectiefuncties ook gebruiken om de volgorde van de argumenten in een n-tupel te veranderen. Bijvoorbeeld:

$$concat(x,y) = concat(\delta_2^2(x,y), \delta_2^1(x,y)) = concat(y,x)$$

We zullen in het vervolg niet steeds projecties in onze definities opnemen, louter om ervoor te zorgen dat de vorm van de definities met de precieze gewenste vorm overeenkomt. We spreken af dat in composities van de vorm

$$h(g_1(x_1,x_2,\ldots,x_n), g_2(x_1,x_2,\ldots,x_n),\ldots,g_m(x_1,x_2,\ldots,x_n))$$

niet elk argument $x_1,x_2,\ldots,x_n$ in iedere functie g_i gebruikt behoeft te worden en verder dat de volgorde van de argumenten er niet toe doet. We staan dus definities toe, zoals

$$f_2(x,y) = concat(concat(y,x), omgek(x)).$$

Een andere manier om nieuwe functies uit oude te construeren is het

gebruiken van recursie. Al in hoofdstuk 1 zagen we hoe krachtig deze methode kan zijn. We zullen nu het gebruik van recursie bij het definiëren van functies op strings formaliseren. Laten we eerst eens een definitie met behulp van recursie bekijken van een unaire functie $f:B\to B$. Als we veronderstellen dat $h_1,h_2:B^2\to B$ al gedefinieerd zijn dan neemt de definitie van f in termen van h_1 en h_2 de volgende vorm aan

$$f(\varepsilon) = w(\text{ofwel voor een } w\in B \text{ ofwel } w \text{ is ongedefinieerd})$$
$$f(0x) = h_1(x,f(x))$$
$$f(1x) = h_2(x,f(x))$$

Het is echter mogelijk dat een of meer argumenten van h_1 en h_2 ontbreken of niet in volgorde staan.

Een recursieve definitie van de identiteitsfunctie $iden:B\to B$ in termen van *cons0* en *cons1* is:

$$iden(\varepsilon) = \varepsilon$$
$$iden(0x) = cons0(x)$$
$$iden(1x) = cons1(x)$$

Een tweede voorbeeld: de functie $len:B\to B$ berekent in unaire vorm de lengte van een string $\in B$. Deze functie wordt als volgt recursief gedefinieerd.

$$len(\varepsilon) = \varepsilon$$
$$len(0x) = cons0(len(x))$$
$$len(1x) = cons1(len(x))$$

We gaan de hierboven gebruikte methode ter definitie van functies nu uitbreiden naar recursieve definities van n-aire functies voor $n\geq 2$. Veronderstel dat g een al gedefinieerde $(n-1)$-aire functie $B^{n-1}\to B$ is en dat h_1 en h_2 al gedefinieerde $(n+1)$-aire functies $B^{n+1}\to B$ zijn. We kunnen nu de functie f als volgt in termen van g, h_1 en h_2 definiëren.

$$f(\varepsilon,x_2,\ldots,x_n) = g(x_2,\ldots,x_n)$$
$$f(0x_1,x_2,\ldots,x_n) = h_1(x_1,x_2,\ldots,x_n,f(x_1,x_2,\ldots,x_n))$$
$$f(1x_1,x_2,\ldots,x_n) = h_2(x_1,x_2,\ldots,x_n,f(x_1,x_2,\ldots,x_n))$$

We staan weer toe dat g een ariteit $<n-1$ heeft en dat h_1 en h_2 ariteiten $<n+1$ hebben. Dit betekent dat het mogelijk is dat één of meer van de argumenten van g, h_1 of h_2 mogen worden weggelaten en dat zij ook niet in volgorde hoeven te staan. Merk op dat de juistheid van een recursieve definitie van f altijd kan worden gecontroleerd met behulp van volledige inductie naar de lengte van het eerste argument van f. We kunnen nu bijvoorbeeld recursie gebruiken om de functie *concat* te definiëren in termen van *iden*, *cons0* en *cons1*.

$$\begin{aligned} concat(\varepsilon,y) &= iden(y) \\ concat(0x,y) &= cons0(concat(x,y)) \\ concat(1x,y) &= cons1(concat(x,y)) \end{aligned}$$

We gebruiken inductie naar de lengte van x om te bewijzen dat $concat(x,y) = xy$ voor alle $x,y \in B$. Als $x = \varepsilon$ dan is $concat(\varepsilon,y) = iden(y) = y$. Veronderstel dat $concat(x,y) = xy$ voor alle strings x met lengte $\leqslant n$. Als we nu een string x met lengte $(n+1)$ bekijken dan is $x = 0z$ of $x = 1$, waarbij $z = staart(x)$ een string ter lengte n is. Als $x = 0z$ dan is $concat(x,y) = cons0(concat(z,y)) = cons0(zy)$ wegens de inductieveronderstelling en $cons0(zy) = 0zy = yx$. In dit geval is dus $concat(x,y) = xy$. Het geval waarin $x = 1z$ kan op overeenkomstige wijze worden behandeld en dus mogen we concluderen dat $concat(x,y) = xy$ ook geldt voor een string x ter lengte $(n+1)$. Hiermee is de eigenschap voor alle $x,y \in B$ met inductie bewezen.

Bij de hierna volgende voorbeelden van recursieve definities zullen we het inductiebewijs niet meer vermelden. We gaan ervan uit dat de lezer dit zelf kan formuleren.

Om de functie *omgek* recursief te definiëren moeten we eerst de functies $append0: B \to B$ en $append1: B \to B$ definiëren:

$$\begin{aligned} append0(x) &= concat(x, nul(x)) \\ append1(x) &= concat(x, een(x)) \end{aligned}$$

Dus

$$\begin{aligned} omgek(\varepsilon) &= \varepsilon \\ omgek(0x) &= append0(omgek(x)) \\ omgek(1x) &= append1(omgek(x)) \end{aligned}$$

Een ander voorbeeld is de complement functie die van een string $x \in B$ alle nullen in enen verandert en omgekeerd. Dus *complement*(0010) = 1101. Een recursieve definitie luidt

$$complement(\varepsilon) = \varepsilon$$
$$complement(0x) = cons1(complement(x))$$
$$complement(1x) = cons0(complement(x))$$

De volgende stelling geeft het belang aan van recursieve definities.

Stelling 5.2

Als f recursief wordt gedefinieerd in termen van Turing-berekenbare functies dan is ook f Turing-berekenbaar.

Bewijs. We geven een schets van het bewijs voor een unaire functie maar een uitbreiding naar het geval waarin f n-air is, $(n \geqslant 2)$ is vrij eenvoudig.

We gaan ervan uit dat f gedefinieerd is in termen van twee Turing-berekenbare functies h_1 en h_2. De definitie luidt

$$f(\varepsilon) = w (w \text{ ongedefinieerd of in } B)$$
$$f(0x) = h_1(x, f(x))$$
$$f(1x) = h_2(x, f(x))$$

De een-tape Turingmachines die h_1 en h_2 berekenen, zullen we M_1 en M_2 noemen. We veronderstellen niet dat bij de definitie van h_1 en h_2 noodzakelijk beide argumenten x en $f(x)$ zijn gebruikt.

De TM, M die f berekent heeft drie tapes. De eerste tape is een input/output-tape, met de tweede simuleren we een stack en de derde gebruiken we als kladblok. Veronderstel dat we de string $z \in B$ als input gebruiken voor M. Als $z = \varepsilon$ en als $w \in B$ dan maakt de machine tape 1 schoon en schrijft w op locaties $1, 2, \ldots, |w|$ van die tape. Als $z = \varepsilon$ maar $w \notin B$ dan geraakt de machine in een eindeloze lus. Als $z \neq \varepsilon$ dan geldt ofwel $z = 0x$ ofwel $z = 1x$ voor een of andere $x \in B$. Als $z = 0x$ en h_1 gebruikt $f(x)$ als argument dan wordt de string z gevolgd door een scheidingsteken / geschreven in de meest linkse niet-lege locaties > 0 van tape 2. Als h_1 echter $f(x)$ niet als

argument gebruikt dan kan de waarde van $f(z)$ direct met behulp van de machine M_1 uit x op tape 3 worden berekend. Op overeenkomstige wijze schrijven we $z/$ op tape 2 als $z = 1x$ en als h_2 $f(x)$ als argument gebruikt en berekenen we anders $f(x)$ met behulp van M_2 op tape 3. In de gevallen waarin we $z/$ naar tape 2 schrijven is de volgende stap het vervangen van z op tape 1 door x. Als $x \neq \varepsilon$ dan herhalen we het proces.

Uiteindelijk komen we in een van de volgende situaties.

Geval 1. ε staat op tape 1 en op tape 2 hebben we $z/staart(z)/\ldots/staart^{|z|-1}(z)/$ geschreven. We beginnen nu met de tweede fase van de berekening. Als $w \in B$ dan coderen we het paar (ε, w) op tape 3. We lezen vervolgens het symbool rechts van het twee /-symbool van rechts op tape 2 om te controleren of $staart^{|z|-1}(z)$ een 0 als eerste symbool heeft. Als dit het geval is dan gebruiken we M_1 om $f(staart^{|z|-1}(z))$ op tape 3 te berekenen; indien niet dan gebruiken we M_2. We vervangen vervolgens $f(staart^{|z|-1}(z))$ door een codering van het paar $(staart^{|z|-1}(z), f(staart^{|z|-1}(z)))$ en we verwijderen de string $staart^{|z|-1}(z)/$ van tape 2. We herhalen nu het proces ter berekening van $f(staart^{|z|-2}(z))$, $f(staart^{|z|-3}(z))$, enzovoort. Als we niet in een oneindige lus terecht komen tijdens de simulaties van M_1 en M_2 dan bereiken we uiteindelijk een situatie waarin $f(z)$ op tape 3 staat en waarin tape 2 de string $z/$ bevat. We kopiëren nu $f(z)$ naar tape 1 en we stoppen. Als echter w niet gedefinieerd is of als er een eindeloze lus optreedt in de simulaties van M_1 of M_2 dan is $f(z)$ niet gedefinieerd en wordt er geen resultaat berekend.

Geval 2. Als tape 2 leeg is en $f(z)$ is gedefinieerd dan moet de waarde van $f(z)$ op tape 3 staan en dan kan deze eenvoudig naar tape 1 worden gekopieerd. Als tape 2 niet leeg is dan staat voor een of andere $|z| > i \geqslant 1$ de string $y = staart^{|z|-i}(z)$ op tape 1, $z/staart(z)/\ldots/staart^{|z|-i-1}(z)/$ op tape 2 en $f(y)$ op tape 3. We gebruiken nu de tweede fase van het berekeningsproces zoals onder geval 1 werd beschreven. We coderen $(y, f(y))$ naar tape 3 en berekenen daar $f(staart^{|z|-i-1}(z))$. We gaan nu verder op dezelfde manier als eerder beschreven en berekenen $f(staart^{|z|-i-2}(z)), \ldots, f(staart(z))$ en tenslotte $f(z)$. Als $f(z)$ gedefinieerd is dan treedt er geen eindeloze lus op en kan de waarde naar tape 1 worden gekopieerd en M kan stoppen.

PRIMITIEF RECURSIEVE FUNCTIES EN PREDICATEN

Primitief recursieve functies op B zijn totale functies $B^n \to N$, $(n \geq 1)$ die worden geconstrueerd volgens de volgende regels.

(1) *Basisfuncties.* Primitief recursieve functies zijn:

(a) $nil: B \to B$ gedefinieerd als

$$nil(x) = \varepsilon \quad \text{voor alle} \quad x \in B;$$

(b) $cons0$ en $cons1: B \to B$, gedefinieerd als

$$cons0(x) = 0x$$

en $cons1(x) = 1x$ voor alle $x \in B$;

(c) de selectorfuncties $\delta^i_n (1 \leq i \leq n)$, waarbij $\delta^i_n : B^n \to B$ is gedefinieerd als

$$\delta^i_n(x_1, x_2, \ldots, x_n) = x_i.$$

(2) Uit bekende primitief recursieve functies kunnen we nieuwe functies construeren door middel van

(a) *compositie*,

(b) *primitieve recursie*, hierbij gebruiken we recursieve definities zoals we hiervoor behandelden, maar we laten geen partiële functies toe. Hiertoe eisen we dat een recursieve definitie van een n-aire functie de volgende vorm heeft:

ofwel (als $n = 1$)

$$\begin{aligned} f(\varepsilon) &= w \\ f(0x) &= h_1(x, f(x)) \\ f(1x) &= h_2(x, f(x)) \end{aligned}$$

met $w \in B$ en h_1, h_2 primitief recursief,

ofwel (als $n > 1$)

$$\begin{aligned} f(\varepsilon, x_1, \ldots, x_n) &= g(x_2, \ldots, x_n) \\ f(0x_1, x_2, \ldots, x_n) &= h_1(x_1, x_2, \ldots, x_n) f(x_1, x_2, \ldots, x_n)) \\ f(1x_1, x_2, \ldots, x_n) &= h_2(x_1, x_2, \ldots, x_n) f(x_1, x_2, \ldots, x_n)) \end{aligned}$$

met g, h_1 en h_2 primitief recursief. In beide bovenstaande gevallen mogen argumenten van g, h_1 of h_2 worden weggelaten en behoeven de argumenten niet in volgorde te staan.

(3) Geen andere functies dan de basisfuncties en de functies die volgens de in (2) gegeven regels kunnen worden geconstrueerd zijn primitief recursief.

Omdat alle basisfuncties totaal zijn en omdat composities en primitieve recursie de totaliteitseigenschap niet verstoren mogen we concluderen dat alle primitief recursieve functies totaal zijn. Verder volgt de volgende stelling uit de stellingen 5.1 en 5.2 omdat de basisfuncties Turing-berekenbaar zijn.

Stelling 5.3

Elke primitief recursieve functie is een Turing-berekenbare totale functie.

Tot nu toe moesten we om aan te tonen dat een functie Turing-berekenbaar was, ofwel een specifieke TM construeren waarmee de functie kon worden berekend, ofwel we moesten onze toevlucht nemen tot een informele argumentatie. Vanaf heden is de situatie aanmerkelijk verbeterd: als we kunnen aantonen dat een functie primitief recursief is dan weten we dat de functie ook Turing-berekenbaar moet zijn. We lieten al zien dat de volgende functies primitief recursief en dus TM-berekenbaar zijn: *nul*, *een*, *iden*, *len*, *concat*, *omgek* en *complement*.

In het vervolg zullen we nog een primitief recursieve functie gebruiken, namelijk de functie *alsleeg*. De expressie $alsleeg(x_1,x_2,x_3)$ levert x_2 als $x_1 = \varepsilon$ en anders x_3. Dus:

$$alsleeg(\varepsilon,x_2,x_3) = iden(x_2)$$
$$alsleeg(0x_1,x_2,x_3) = iden(x_3)$$
$$alsleeg(1x_1,x_2,x_3) = iden(x_3)$$

Op overeenkomstige wijze definiëren we de functies *alsnul* en *alseen*.

$$\begin{aligned} alsnul(\varepsilon, x_2, x_3) &= iden(x_3) \\ alsnul(0x_1, x_2, x_3) &= alsleeg(x_1, x_2, x_3) \\ alsnul(1x_1, x_2, x_3) &= iden(x_3) \end{aligned}$$

en

$$\begin{aligned} alseen(\varepsilon, x_2, x_3) &= iden(x_3) \\ alseen(0x_1, x_2, x_3) &= iden(x_3) \\ alseen(1x_1, x_2, x_3) &= alsleeg(x_1, x_2, x_3) \end{aligned}$$

Andere belangrijke primitieve recursieve functies zijn de functies *kop*, *staart*, *voet* en *top* die we allen al eerder behandelden. De functie *kop* wordt met behulp van primitieve recursie als volgt gedefinieerd

$$\begin{aligned} kop(\varepsilon) &= \varepsilon \\ kop(0x) &= nul(x) \\ kop(1x) &= een(x) \end{aligned}$$

Evenzo definiëren we *staart* met

$$\begin{aligned} staart(\varepsilon) &= \varepsilon \\ staart(0x) &= iden(x) \\ staart(1x) &= iden(x) \end{aligned}$$

Het definiëren van *voet* en *top* door middel van primitieve recursie laten we als oefening aan de lezer over (oefening 5.4).

Als een functie $B^n \to B$ als codomein $\{0,1\}$ heeft (te interpreteren als respectievelijk *onwaar* en *waar*) dan noemen we deze functie een n-air *predicaat*. In het navolgende zullen primitief recursieve predicaten een belangrijke rol spelen. Twee predicaten die we al tegenkwamen zijn *nul* en *een*; deze functies zijn beide unair. Ook de functie *isnul* is een unair primitief recursief predicaat, gedefinieerd door

$$isnul(x) = \begin{cases} 1 & \text{als} \quad x = \varepsilon \\ 0 & \text{anders} \end{cases}$$

De functie *isnul* kan als volgt primitief recursief worden gedefinieerd

$$\begin{aligned} isnul(\varepsilon) &= 1 \\ isnul(0x) &= nul(x) \\ isnul(1x) &= nul(x) \end{aligned}$$

Vervolgens wordt het unaire primitief recursieve predicaat *iswaar*(x), dat test of $x = 1$ gedefinieerd door

$$\begin{aligned} iswaar(\varepsilon) &= 0 \\ iswaar(0x) &= nul(x) \\ iswaar(1x) &= isnul(x) \end{aligned}$$

Als $P(x_1,x_2,\ldots,x_n)$ een n-air predicaat is dan kunnen we $\sim P(x_1,x_2,\ldots,x_n)$, de *ontkenning* van P, uit P construeren door middel van de definitie

$$\sim P(x_1,x_2,\ldots,x_n) = \begin{cases} 1 & \text{als } P(x_1,x_2,\ldots,x_n) = 0 \\ 0 & \text{als } P(x_1,x_2,\ldots,x_n) = 1 \end{cases}$$

Als $P(x_1,x_2,\ldots,x_n)$ primitief recursief is dan is $\sim P(x_1,x_2,\ldots,x_n)$ dat ook omdat

$$\sim P(x_1,x_2,\ldots,x_n) = alseen(P(x_1,x_2,\ldots,x_n), nul(x_1), een(x_1))$$

Als $P_1(x_1,x_2,\ldots,x_n)$ en $P_2(x_1,x_2,\ldots,x_n)$ n-aire predicaten zijn dan kunnen we met behulp van de connectieven $\wedge$ (en), $\vee$ (of) en $\Rightarrow$ (impliceert) als volgt nieuwe predicaten construeren.

$$P_1(x_1,x_2,\ldots,x_n) \wedge P_2(x_1,x_2,\ldots,x_n) = \begin{cases} 1 & \text{als } P_1(x_1,x_2,\ldots,x_n) = 1 \\ & \text{en } P_2(x_1,x_2,\ldots,x_n) = 1 \\ 0 & \text{anders} \end{cases}$$

$$P_1(x_1,x_2,\ldots,x_n) \vee P_2(x_1,x_2,\ldots,x_n) = \begin{cases} 1 & \text{als } P_1(x_1,x_2,\ldots,x_n) = 1 \\ & \text{of } P_2(x_1,x_2,\ldots,x_n) = 1 \\ 0 & \text{anders} \end{cases}$$

$$P_1(x_1,x_2,\ldots,x_n) \Rightarrow P_2(x_1,x_2,\ldots,x_n) = \begin{cases} 1 & \text{als ofwel} \\ & \text{ofwel } P_1(x_1,x_2,\ldots,x_n) = 0 \\ & \text{ofwel} \\ & \text{zowel } P_1(x_1,x_2,\ldots,x_n) = 1 \\ & \text{alsook } P_2(x_1,x_2,\ldots,x_n) = 1 \\ 0 & \text{anders} \end{cases}$$

Als we nieuwe predicaten $(P_1 \wedge P_2)$, $(P_1 \vee P_2)$, $(P_1 \Rightarrow P_2)$ definiëren door

$$(P_1 \wedge P_2)(x_1,x_2,\ldots,x_n) = P_1(x_1,x_2,\ldots,x_n) \wedge P_2(x_1,x_2,\ldots,x_n)$$
$$(P_1 \vee P_2)(x_1,x_2,\ldots,x_n) = P_1(x_1,x_2,\ldots,x_n) \vee P_2(x_1,x_2,\ldots,x_n)$$
$$(P_1 \Rightarrow P_2)(x_1,x_2,\ldots,x_n) = P_1(x_1,x_2,\ldots,x_n) \Rightarrow P_2(x_1,x_2,\ldots,x_n)$$

dan zijn $(P_1 \wedge P_2)$, $(P_1 \vee P_2)$ en $(P_1 \Rightarrow P_2)$ primitief recursief als P_1 en P_2 dat zijn. Immers

$$(P_1 \wedge P_2)(x_1,x_2,\ldots,x_n) = alseen(P_1(x_1,x_2,\ldots,x_n), P_2(x_1,x_2,\ldots,x_n), nul(x_1))$$
$$(P_1 \vee P_2)(x_1,x_2,\ldots,x_n) = alseen(P_1(x_1,x_2,\ldots,x_n), een(x_1), P_2(x_1,x_2,\ldots,x_n))$$
$$(P_1 \Rightarrow P_2)(x_1,x_2,\ldots,x_n) = alseen(P_1(x_1,x_2,\ldots,x_n), P_2(x_1,x_2,\ldots,x_n), een(x_1))$$

Om dit aan te tonen hadden we ook kunnen gebruiken dat

$$(P_1 \wedge P_2)(x_1,x_2,\ldots,x_n) = \sim P_1(x_1,x_2,\ldots,x_n) \vee P_2(x_1,x_2,\ldots,x_n).$$

In hoofdstuk 2 behandelden we enkele berekenbare rekenkundige bewerkingen met behulp van de unaire of de binaire voorstelling van natuurlijke getallen. We definiëren nu een predicaat $unair(x)$ dat als resultaat 1 oplevert als $x \in B$. Een toegelaten unaire voorstelling is van een natuurlijk getal en dat 0 oplevert als dat niet het geval is. Dit predicaat is primitief recursief omdat

$$unair(\varepsilon) = 0$$
$$unair(0x) = nul(x)$$
$$unair(1x) = unair(x) \vee isnul(x)$$

Als $unair(x) = 1$ dan zullen we met $\bar{x}$ het gehele getal weergeven dat door x in unaire notatie wordt voorgesteld.

Laten we nu eens onderzoeken hoe we functies kunnen definiëren die eenvoudige rekenkundige bewerkingen uitvoeren met behulp van unaire representaties. We zorgen ervoor dat de functies totaal zijn door 0 voor ongedefinieerd te gebruiken. Bijvoorbeeld

$$plus(x,y) = \begin{cases} 0 & \text{als } \sim unair(x) \vee \sim unair(y) \\ z & \text{anders, met } \bar{z} = \bar{x} + \bar{y} \end{cases}$$

Deze operatie is primitief recursief omdat

$$plus(x,y) = alseen(unair(x) \wedge unair(y), concat(x,y), nul(x)))$$

In oefening 5.5 wordt de lezer verzocht aan te tonen dat ook de bewerkingen *minus*, *maal* en *gedeelddoor* primitief recursief zijn. Een rekenkundig bewerking van bijzonder belang is de functie *machta*(n) die een getal a tot een gehele macht n verheft en dus a^n berekent. Als we veronderstellen dat getallen unair worden weergegeven dan kunnen we bijvoorbeeld definiëren

$$macht2(x) = \begin{cases} 0 & \text{als } \sim unair(x) \\ y & \text{anders, waarbij } \bar{y} = 2^{\bar{x}} \end{cases}$$

Deze operatie is primitief recursief omdat

$$macht2(x) = alseen(unair(x), umacht2(x), nul(x)))$$

met

$$umacht2(\varepsilon) = 1$$
$$umacht2(0x) = nul(x) \quad \text{(of een willekeurige andere functie omdat dit niet kan voorkomen)}$$
$$umacht2(1x) = concat(umacht2(x), umacht2(x))$$

Behalve *unair* definiëren we een predicaat *binair* om na te gaan of een string $x \in B$ een toegelaten binaire representatie is van een natuurlijk getal.

$$binair(\varepsilon) = 0$$
$$binair(0x) = nul(x)$$
$$binair(1x) = een(x)$$

Conversie van binair naar unair wordt uitgevoerd met de functie $binun: B \rightarrow B$. Als $x \in B$ een toegelaten binaire representatie is dan geeft $binun(x)$ de bijbehorende unaire weergave; in alle andere gevallen is het resultaat 0.

$$binun(x) = alseen(binair(x), binun1(x), nul(x))$$

met

$$\begin{aligned} binun1(\varepsilon) &= \varepsilon \\ binun1(0x) &= binun1(x) \\ binun1(1x) &= concat(umacht2(len(x)), binun1(x)) \end{aligned}$$

We laten het aan de lezer over om aan te tonen dat de omgekeerde conversie van unair naar binair met behulp van een functie *unbin* eveneens primitief recursief is, (oefening 5.6). Als we nu een n-aire rekenkundige functie f_1 hebben die uitgaat van een unaire representatie en als we hebben aangetoond dat die functie primitief recursief is, dan volgt hieruit dat een overeenkomstige functie f_2 die uitgaat van een binaire representatie, eveneens primitief recursief is.

Op dezelfde wijze kunnen we concluderen als we een rekenkundige functie hebben die een binaire representatie veronderstelt en als we hebben aangetoond dat die primitief recursief is, dat ook de bijbehorende functie die uitgaat van unaire representatie primitief recursief is.

Als x en y de unaire voorstellingen zijn van de natuurlijke getallen $\bar{x}$ en $\bar{y}$ en als $\bar{x} < \bar{y}$ dan levert het primitief recursieve predicaat $kleinerdan(x,y)$ de waarde 1 op; in alle andere gevallen is de waarde 0.

$$kleinerdan(x,y) = alseen(unair(x) \quad unair(y), kleiner(x,y), nul(x))$$

met

$$\begin{aligned} kleiner(\varepsilon,y) &= \sim isnul(y) \\ kleiner(0x,y) &= nil(x) \quad \text{(of een willekeurige andere functie omdat dit geval niet kan optreden)} \\ kleiner(1x,y) &= alsleeg(y, nul(x), kleiner(x, staart(y))) \end{aligned}$$

De natuurlijke lexicografische ordening van $B: \varepsilon, 0, 1, 00, 01, \ldots$ zijn we al herhaalde malen tegengekomen en dit zal ook in het vervolg van dit boek nog vaak het geval zijn. Het is daarom handig om wat primitief recursieve functies en predicaten te hebben in verband met deze ordening. De functie *volgend* levert bij een input $x \in B$ de volgende string uitgaande van de lexicografische ordening. De functie is gedefinieerd als

$$volgend(x) = omgek(voorvolgend(omgek(x)))$$

met

$$voorvolgend(\varepsilon) = 0$$
$$voorvolgend(0x) = cons1(x)$$
$$voorvolgend(1x) = cons0(voorvolgend(x))$$

De voorganger functie *vorig* levert bij een input $x \in B$ de voorganger van de string x uitgaande van lexicografische ordening. We zorgen ervoor dat *vorig* totaal is door $vorig(\varepsilon) = \varepsilon$ te definiëren.

$$vorig(x) = omgek(voorvorig(omgek(x))$$

met

$$voorvorig(\varepsilon) = \varepsilon$$
$$voorvorig(0x) = alsleeg(x, nil(x), cons1(voorvorig(x)))$$
$$voorvorig(1x) = cons0(x)$$

Als x de i-de string is in de lexicografische ordening dan geeft $pos(x)$ als resultaat de unaire representatie van i.

$$pos(x) = concat(umacht2(len(x)), binun1(x))$$

Omgekeerd als voor $x \in B$ geldt dat $unair(x)$ waar is dan levert $lex(x)$ een string y, zodanig dat $pos(y) = x$; in alle andere gevallen levert $lex(x)$ ε als resultaat.

$$lex(x) = alseen(unair(x), len2(x), nil(x))$$

met

$lex2(\varepsilon) = \varepsilon$ (of een willekeurige andere waarde)
$lex2(0x) = nil(x)$ (of een willekeurige andere functie)
$lex2(1x) = alsleeg(x, nil(x), volgend(len2(x)))$

We kunnen nu ook bewijzen dat het predicaat $voorganger(x,y)$

$$voorganger(x,y) = \begin{cases} 1 & \text{als } x \text{ voor } y \text{ komt op grond van de lexicografische ordening} \\ 0 & \text{anders} \end{cases}$$

primitief recursief is. Immers

$$voorganger(x,y) = kleiner(pos(x),pos(y))$$

Het gelijkheidspredicaat $isgelijk(x,y)$ dat 1 levert als $x = y$ en 0 in alle andere gevallen is ook primitief recursief omdat

$$isgelijk(x,y) = \sim voorganger(x,y) \wedge \sim voorganger(y,x).$$

Nog een voorbeeld van een primitief recursief predicaat is de functie *even*, met $even(x) = 1$ desd als x een binaire representatie is van een even getal.

$$even(x) = binair(x) \wedge eindigtop0(x)$$

met

$$eindigtop0(\varepsilon) = 0$$
$$eindigtop0(0x) = eindigtop0(x) \vee isnul(x)$$
$$eindigtop0(1x) = eindigtop0(x)$$

De functie *oneven*, die 1 levert als x oneven is wordt op overeenkomstige wijze gedefinieerd.

$$oneven(x) = binair(x) \wedge eindigtop1(x)$$

met

$$eindigtop1(\varepsilon) = 0$$
$$eindigtop1(0x) = eindigtop1(x)$$
$$eindigtop1(1x) = eindigtop1(x) \vee isnul(x)$$

Zo zouden we door kunnen gaan met het definiëren van primitief recursieve functies en predicaten. Het is een grote en belangrijke klasse functies, maar omdat al deze functies totaal zijn betreft het hier niet de klasse van alle Turing-berekenbare functies. Gaat het dan tenminste om alle totale Turing-berekenbare functies? Zelfs dat blijkt helaas niet waar te zijn! Toch vormen de primitief recursieve functies een zeer belangrijke deelverzameling van alle totale berekenbare functies $B^n \rightarrow B$, omdat compositie en primitieve recursie de meest elementaire methoden voor het construeren van functies zijn. Het is trouwens niet eenvoudig een totale functie $B^n \rightarrow B$ te construeren die Turing-

berekenbaar is, maar die niet primitief recursief is. Een zeer bekend voorbeeld is *Ackermann's functie* $Ack: B \to B$.

$$Ack(x) = a(x,x)$$

met

$$a(x_1,x_2) = \begin{cases} 0x_2 & \text{als } x_1 = \varepsilon \\ a(staart(x_1),0) & \text{als } x_1 \neq \varepsilon \text{ en } x_2 = \varepsilon \\ a(staart(x_1),a(x_1,staart(x_2))) & \text{als } x_1 \neq \varepsilon \text{ en } x_2 \neq \varepsilon \end{cases}$$

Om te bewijzen dat Ack niet primitief recursief is moet men aantonen dat voor iedere unaire primitieve functie g er een $x \in B$ bestaat, zodanig dat $|Ack(x)| > |g(x)|$. (Zie bijvoorbeeld Hermes, 1965.)

PARTIEEL RECURSIEVE FUNCTIES

Voordat we de klasse van partieel recursieve functies kunnen definiëren moeten we nog een nieuwe manier behandelen waarop nieuwe functies uit oude kunnen worden geconstrueerd. Het gaat hier om de methode van *onbegrensde minimalisatie*, (zie oefening 5.8 voor het verschil met begrensde minimalisatie).

Als $P(x_1,x_2,\ldots,x_n)$ een n-air primitief recursief predicaat is ($n \geqslant 2$) dan construeren we als volgt een (n-1)-aire functie $f(x_1,x_2,\ldots,x_{n-1})$ uit $P(x_1,x_2,\ldots,x_n)$ met behulp van onbegrensde minimalisatie. We definiëren de waarde van $f(x_1,x_2,\ldots,x_{n-1})$ als de eerste string y in de natuurlijke lexicografische ordening van B, zodanig dat $P(x_1,x_2,\ldots,x_{n-1},y) = 1$. Als zo'n y niet bestaat dan is f ongedefinieerd. De notatie voor deze definitiewijze is

$$f(x_1,x_2,\ldots,x_{n-1}) = \mu y \cdot P(x_1,x_2,\ldots,x_{n-1},y)$$

en we lezen $\mu y \cdot P(x_1,x_2,\ldots,x_{n-1},y)$ als 'de kleinste y, waarvoor $P(x_1,x_2,\ldots,x_{n-1},y)$ waar is'.

Stelling 5.4

Als P een n-air primitief recursief predicaat is ($n \geqslant 2$) en als f een (n-1)-aire functie is, gedefinieerd met behulp van onbegrensde minimalisatie dan is f Turing-berekenbaar.

Bewijs. Omdat P primitief recursief is mogen we veronderstellen dat P wordt berekend door een Turingmachine M. De TM die f berekent heeft nu drie tapes. De eerste tape wordt alleen voor input/output gebruikt en op de tweede wordt B in lexicografische ordening $\varepsilon,0,1,00,01,\ldots$ gegenereerd. Zodra een string y uit deze ordening wordt gegenereerd simuleren we op tape 3 het gedrag van de machine M met input $x_1,x_2,\ldots,x_{n-1},y$, waarbij $x_1,x_2,\ldots,x_{n-1}$ van tape 1 wordt gekopieerd en y van tape 2. De simulatie is altijd eindig omdat P een totale berekenbare functie is. Als $P(x_1,x_2,\ldots,x_{n-1},y)$ de waarde 1 blijkt te hebben dan wordt tape 1 schoongemaakt, wordt y naar tape 1 gekopieerd en stopt de berekening. In alle andere gevallen maken we tape 3 schoon, genereren de volgende string uit de lexicografische ordening op tape 2 en vervolgen het zoekproces.

We noemen een partiële functie $f:B^n \to B$ een *partieel predicaat* als $f(x_1,x_2,\ldots,x_n)$ ofwel niet is gedefinieerd, ofwel de waarde 0 of 1 heeft. Bovenstaand bewijs is niet geldig voor TM-berekenbare partiële predicaten omdat het niet zeker is dat de simulatie op tape 3 altijd zal stoppen. Het kan voorkomen dat $P(x_1,x_2,\ldots,x_{n-1},y)$ maar dat er een y' bestaat die voorafgaat aan y, waarvoor $P(x_1,x_2,\ldots,x_{n-1},y')$ ongedefinieerd is. Als deze situatie echter niet voorkomt dan kunnen we ons bewijs generaliseren. We definiëren daarom als $P(x_1,x_2,\ldots,x_n)$ een partieel predicaat is ($n \geqslant 2$)

$$\mu y \cdot P(x_1,x_2,\ldots,x_{n-1},y) = \begin{cases} y' & \text{als } P(x_1,x_2,\ldots,x_{n-1},y') = 1 \\ & \text{en } P(x_1,x_2,\ldots,x_{n-1},y'') = 0 \\ & \text{voor alle voorgangers } y'' \text{ van } y' \\ \text{ongedefinieerd} & \text{anders} \end{cases}$$

Hieruit volgt:

Stelling 5.5

Als P een n-air predicaat is (totaal of partieel) en als P TM-berekenbaar is en als f de $(n-1)$-aire functie is, gedefinieerd door $\mu y \cdot P(x_1,x_2,\ldots,x_{n-1},y)$ dan is f ook TM-berekenbaar.

Een eenvoudig voorbeeld van een partiële recursieve functie, gedefinieerd door onbegrensde minimalisatie, is de functie *ondef*, die altijd ongedefinieerd is.

$$ondef(x) = \mu y \cdot nul(\delta_2^2(x,y))$$

Ook hier laten we de selectorfuncties gewoonlijk weg uit de definities. We mogen dan schrijven

$$ondef(x) = \mu y \cdot nul(y)$$

We kunnen deze functie ook door middel van recursie definiëren omdat

$$\begin{aligned} ondef(\varepsilon) &= \text{ongedefinieerd} \\ ondef(0x) &= ondef(x) \\ ondef(1x) &= ondef(x) \end{aligned}$$

Dit laatste is niet altijd mogelijk: er bestaan partiële functies die we met behulp van onbegrensde minimalisatie kunnen definiëren, maar die niet louter met recursie en compositie kunnen worden geconstrueerd. We kunnen echter eisen dat we bij toepassen van de minimalisatiemethode altijd uitgaan van een primitief recursief predicaat, (zie oefening 5.10).

We noemen een string $x \in B$ een *palindroom* desd als $x = omgek(x)$. Als een palindroom van even lengte is dan kan het geschreven worden als $concat(y,omgek(y))$. De functie *halfpalin* wordt dan gedefinieerd als

$$halfpalin(x) = \begin{cases} y & \text{als } x = concat(y,omgek(y)) \\ \text{ongedefinieerd} & \text{anders} \end{cases}$$

Deze functie is partieel recursief omdat

$$halfpalin(x) = \mu y \cdot P(x,y)$$

Hierbij is P het primitief recursieve predicaat gedefinieerd door

$$P(x,y) = isgelijk(x,concat(y,omgek(y)))$$

De klasse van *partieel recursieve functies* op B bestaat uit functies $B^n \to B$ $(n \geqslant 1)$, geconstrueerd volgens de volgende regels.

(1) *Basisfuncties.* De volgende functies zijn partieel recursief:

(a) $nil: B \to B$ gedefinieerd als

$$nil(x) = \varepsilon \quad \text{voor alle } x \in B$$

(b) $cons0$ en $cons1: B \to B$ gedefinieerd als

$$cons0(x) = 0x$$
$$cons1(x) = 1x \text{ voor alle } x \in B$$

(c) de selectorfuncties $\delta_n^i (1 \leqslant i \leqslant n)$, waarbij $\delta_n^i : B^n \to B$ gedefinieerd wordt als

$$\delta_n^i(x_1,x_2,\ldots,x_n) = x_i$$

(2) Uit bekende partieel recursieve functies en predicaten kunnen we nieuwe partieel recursieve functies construeren door middel van

(a) compositie

(b) recursie

(c) onbegrensde minimalisatie.

Als we recursie gebruiken dan mag de definitie van een n-aire functie de volgende vorm aannemen:

ofwel (geval $n = 1$)

$$f(\varepsilon) = w \quad (\text{of } w \in B \text{ of } w \text{ is ongedefinieerd})$$
$$f(0x) = h_1(x,f(x))$$
$$f(1x) = h_2(x,f(x))$$

met h_1 en h_2 partieel recursief,

ofwel (geval $n > 1$)

$$f(\varepsilon,x_2,\ldots,x_n) = g(x_2,\ldots,x_n)$$
$$f(0x_1,x_2,\ldots,x_n) = h_1(x_1,x_2,\ldots,x_n,f(x_1,x_2,\ldots,x_n))$$
$$f(1x_1,x_2,\ldots,x_n) = h_1(x_1,x_2,\ldots,x_n,f(x_1,x_2,\ldots,x_n))$$

met g, h_1 en h_2 partieel recursief. In beide gevallen mogen argumenten van g, h_1 of h_2 worden weggelaten en/of in een andere volgorde staan.

Als we de methode van de onbegrensde minimalisatie gebruiken dan mag het predicaat elk willekeurig partieel recursief predicaat zijn. Zo-als we al eerder zagen zou de klasse van functies niet veranderen als we ons zouden beperken tot primitief recursieve predicaten.

(3) Behalve de basisfuncties en de functies die kunnen worden geconstrueerd volgens de in (2) gegeven regels bestaan er geen partieel recursieve functies.

Uit deze definitie en de stellingen 5.1, 5.2 en 5.3 volgt onmiddellijk:

Stelling 5.6

(1) Elke partieel recursieve functie op B is Turing-berekenbaar.

(2) Elke partieel recursieve functie op B is partieel recursief op B.

Omdat partieel recursieve functies niet noodzakelijk totaal zijn bestaan er natuurlijk ook partieel recursieve functies die niet primitief recursief zijn. Een voor de hand liggend voorbeeld is de functie $ondef{:}B \to B$.

Na deze voorbereidingen kunnen we nu de belangrijkste stelling uit dit hoofdstuk behandelen.

Stelling 5.7 (Stelling van Kleene)

De partieel recursieve functies op B zijn juist de Turing-berekenbare functies op B.

Bewijs. Uit stelling 5.6(1) volgt dat iedere partieel recursieve functie op B Turing-berekenbaar is. We geven nu een schets van het bewijs dat elke Turing-berekenbare functie op B ook partieel recursief is.

Op grond van stelling 2.6 mogen we veronderstellen dat een TM, M die een (mogelijk partiële) functie $B^n \to B$ berekent, als tape-alfabet $\{0,1,\wedge\}$ heeft. Om te beginnen is het n-tupel $(x_1,x_2,\ldots,x_n) \in B^n$

gecodeerd als een binaire string z en we mogen aannemen dat deze codering zelf partieel recursief is. Zonder verlies van algemeenheid mogen we ook aannemen dat de toestanden van M als labels de natuurlijke getallen $1,2,\ldots,n$ hebben. Hierbij is 1 de begintoestand en zijn alle toestanden k voor een of andere $k \leqslant n$ eindtoestanden. Tenslotte mogen we veronderstellen dat de lees/schrijfkop steeds naar links of naar rechts beweegt en dus nooit blijft stilstaan boven dezelfde positie op de tape.

Laten we nu eens een configuratie $C = (q, i, \alpha, a, \beta)$ van M bekijken met $q \in \{1,2,\ldots,n\}$, $i \in Z, \alpha, \beta \in B$ en $a \in \{\wedge, 0, 1\}$. Zo'n configuratie kunnen we met behulp van een Gödel-nummering coderen als een geheel getal $\hat{C}$. Een mogelijke codering is bijvoorbeeld

$$\hat{C} = \hat{q} \cdot \hat{\imath} \cdot \hat{\alpha} \cdot \hat{a} \cdot \hat{\beta}$$

met

$$\hat{q} = 2^q$$

$$\hat{\imath} = \begin{cases} 3 \cdot 5^i & \text{als } i \geqslant 0 \\ 3^2 \cdot 5^{-i} & \text{als } i < 0 \end{cases}$$

$$\hat{a} = \begin{cases} 7 \cdot 11 & \text{als } \alpha = \varepsilon, \\ 7^2 \cdot 11^{n(\alpha)} & \text{als } kop(x) = 1 \quad \text{en } n(\alpha) \text{ is het gehele getal} \\ & \qquad\qquad\qquad\quad \text{met binaire voorstelling } \alpha \\ 7^3 \cdot 11^{n(\bar{\alpha})} & \text{als } kop(\alpha) = 0, \quad \bar{\alpha} = complement(\alpha) \text{ en } n(\bar{\alpha}) \\ & \qquad\qquad\qquad\quad \text{is het bijbehorende gehele getal} \end{cases}$$

$$\hat{a} = \begin{cases} 13 & \text{als } a = \wedge, \\ 13^2 & \text{als } a = 0, \\ 13^3 & \text{als } a = 1; \end{cases}$$

en
$$\hat{\beta} = \begin{cases} 17 & \text{als } \beta = \varepsilon, \\ 17^2 \cdot 19^{n(\beta)} & \text{als } kop(\beta) = 1, \\ 17^3 \cdot 19^{n(\bar{\beta})} & \text{als } kop(\beta) = 0. \end{cases}$$

Een beginconfiguratie $C_0 = (1, 1, \varepsilon, 0, 101)$ zou dus worden gecodeerd als $C_0 = 2 \cdot 3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13^2 \cdot 17^2 \cdot 19^2$. Het voordeel van het gebruik van Gödel-nummering is dat uit het codegetal C de componenten van C snel en efficiënt kunnen worden teruggevonden door slechts gebruik te maken van eenvoudige rekenkundige bewerkingen. De binaire equivalenten

van deze eenvoudige bewerkingen zijn alle primitief recursieve functies. Laten we nu de binaire voorstelling van een positief geheel getal n weergeven door $bin(n)$. Elke configuratie C kan dan worden voorgesteld door een unieke string $c = bin(\hat{C}) \in B$. We kunnen nu aantonen dat de volgende functies partieel recursief zijn.

$$eerste(x) = \begin{cases} bin(q) & \text{als } x = bin(\hat{C}) \text{ voor een configuratie } C \text{ met als eerste component } q \\ \text{niet gedefinieerd} & \text{anders} \end{cases}$$

$$tweede(x) = \begin{cases} bin(i) & \text{als } x = bin(\hat{C}) \text{ voor een configuratie } C \text{ met als tweede component } i \\ \text{niet gedefinieerd} & \text{anders} \end{cases}$$

$$derde(x) = \begin{cases} \alpha & \text{als } x = bin(\hat{C}) \text{ voor een configuratie } C \text{ met als derde component } \alpha \\ \text{niet gedefinieerd} & \text{anders} \end{cases}$$

$$vierde(x) = \begin{cases} a & \text{als } x = bin(\hat{C}) \text{ voor een configuratie } C \text{ met als vierde component } a \\ \text{niet gedefinieerd} & \text{anders} \end{cases}$$

$$vijfde(x) = \begin{cases} \beta & \text{als } x = bin(\hat{C}) \text{ voor een configuratie } C \text{ met als vijfde component } \beta \\ \text{niet gedefinieerd} & \text{anders} \end{cases}$$

Laten we nu een TM, $M = (\{1,2,\ldots,n\},\{0,1,\wedge\},\{0,1\},P,1,\{k,\ldots,n\})$ bekijken. De functie $P: \{1,2,\ldots,k-1\} \times \{0,1\} \to \{1,2,\ldots,n\} \times \{0,1\} \times \{L,R\}$ kan dan worden voorgesteld door middel van drie functies.

$volgendetoestand: \{1,2,\ldots,k-1\} \times \{0,1\} \to \{1,2,\ldots,n\}$
$volgendesymbool: \{1,2,\ldots,k-1\} \times \{0,1\} \to \{0,1\}$
$stap: \{1,2,\ldots,k-1\} \times \{0,1\} \to \{L,R\}$

met $P(i,b) = (volgendetoestand(i,b), volgendesymbool(i,b), stap(i,b))$ $i \in \{1,2,\ldots,k-1\}$ en $b \in \{0,1\}$. Functies $B^2 \to B$ die overeenkomen met

volgendetoestand, *volgendesymbool* en *stap* kunnen we construeren door de getallen $1,2,\ldots,n$ binair weer te geven en door 0 voor L en 1 voor R te gebruiken. Deze binaire functies noemen we *bvolgendetoestand*, *bvolgendesymbool* en *bstap*. Deze drie functies zijn alle partieel recursief want zij kunnen van geval tot geval worden gedefinieerd, (zie oefening 5.9).

We kunnen nu partieel recursieve functies $B \to B$ definiëren die, gegeven een binaire voorstelling van een getal $n \geqslant 0$, de binaire voorstelling van 2^n, 3^n, 5^n, 7^n, enzovoorts berekenen. Er is hier sprake van partiële functies omdat de ongedefinieerd zijn als de inputstring met een 0 begint. We noemen deze functies *bmacht2*, *bmacht3*, enzovoorts. De functie $bmult{:}B^2 \to B$ is de vermenigvuldingsfunctie, uitgaande van binaire voorstelling van het argument.

Als we nu de string $z \in B$ als input aan onze Turingmachine geven dan is de beginconfiguratie

$$C_0 = (1,1,\varepsilon,\ dan\ z = \varepsilon\ dan\ \wedge\ anders\ kop(z), staart(z))$$

Deze beginconfiguratie wordt gecodeerd als $c_0 = bin(\hat{C}_0)$ en $c_0 = input(z)$. Hierbij is *input* een partieel recursieve functie gedefinieerd als

$$input(z) = bmult(const(z), bmult(arg4(z), arg5(z)))$$

met

(i) $const(z)$ is een constante functie die als resultaat de binaire string oplevert voor de geheeltallig codering van de eerste drie argumenten $1,1,\varepsilon$ van C_0, dat wil zeggen het getal $2^1{\cdot}3{\cdot}5^2{\cdot}7{\cdot}11$,

(ii) $arg4(z) = bmacht13(alsleeg(z, een(z), alsnul(kop(z), twee(z), drie(z))))$

twee en drie zijn constante functies die respectievelijk de binaire strings 10 en 11 opleveren.

(iii) $arg5(z) = alsleeg(staart(z), bmacht17(een(z)), alseen(kop(staart(z)),$
$bmult(bmacht17(twee(z)), bmacht19(staart(z))),$
$bmult(bmacht17(drie(z)),$
$bmacht19(complement(staart(z))))))$

We zullen nu laten zien dat de functie *volgendeconfiguratie* partieel recursief is. Gegeven een codering van een configuratie C_n van M moet *volgendeconfiguratie* de codering berekenen van de volgende configuratie C_{n+1} onder de voorwaarde dat voor M een volgende stap mogelijk is. Als $C_n = (q, i, \alpha, a, \beta)$ dan

$$C_{n+1} = \begin{cases} (\mathit{volgendetoestand}(q,a), i-1, \mathit{top}(\alpha), \mathit{als}\ \alpha=\varepsilon\ \mathit{dan}\ \wedge\ \mathit{anders}\ \mathit{voet}(\alpha), \\ \quad \mathit{concat}(\mathit{volgendesymbool}(q,a),\beta)) & \text{als } \mathit{stap}(q,a) = L \\ (\mathit{volgendetoestand}(q,a), i+1, \mathit{concat}(\alpha, \mathit{volgendesymbool}(q,a)), \\ \quad \mathit{als}\ \beta=\varepsilon\ \mathit{dan}\ \wedge\ \mathit{anders}\ \mathit{kop}(\beta), \mathit{staart}(\beta)) & \text{als } \mathit{stap}(q,a) = R \\ \text{niet gedefinieerd} & \text{anders} \end{cases}$$

Als nu C_n gecodeerd is als een binaire string $c_n = bin(\hat{C}_n)$ dan kunnen binaire strings voor de componenten q, i, α, a, β worden verkregen door het toepassen van de partiële recursieve functies *eerste*, *tweede*, *derde*, *vierde* en *vijfde*. Vervolgens kunnen we vanuit de definitie van C_{n+1} de binaire $c_{n+1} = bin(\hat{C}_{n+1})$ verkrijgen door deze functies toe te passen samen met de functies *bvolgendetoestand*, *bvolgendesymbool* en *bstap*, de machtfuncties *bmacht2*, *bmacht3*, enzovoorts, de constante functies *alsleeg*, *alsnul*, *alseen* en de functies *kop*, *staart*, *top*, *voet* en *concat*. De uitwerking laten we verder aan de lezer over. Die is niet moeilijk maar wel erg bewerkelijk! Zo krijgen we de partiële functie

$$\mathit{volgendeconfiguratie}(x) = \begin{cases} \text{niet gedefinieerd} & \text{als } x \text{ geen codering is van een toegelaten configuratie van } M\text{, of zelfs als dat wel het geval is, als er geen volgende stap mogelijk is,} \\ y & \text{als } x \text{ de codering is van een configuratie } C_n \text{ en als de volgende stap van } M \text{ tot een configuratie } C_{n+1} \text{ leidt met codering } y. \end{cases}$$

De functie $constant:B \to B$ is de constante functie die met elke binaire string als input als resultaat de unaire representatie van k oplevert. Als x de codering van een configuratie voorstelt dan kunnen we vaststellen of de machine zich in een eindtoestand bevindt door gebruik te maken van het predicaat

$$halt(x) = \sim kleinerdan(binun(eerste(x)), constantk(x))$$

Als $halt(x) = 1$ dan is x de codering van een eindconfiguratie en de bijbehorende output volgt uit $output(x)$. Het is niet moeilijk aan te tonen dat ook deze functie partieel recursief is door gebruik te maken van de definitie van de functie *output* zoals die in hoofdstuk 2 werd gegeven, samen met de functies *eerste*, *tweede*, enzovoorts.

Als $x, y \in B$ dan kunnen we *machinestap* recursief als volgt definiëren

$$machinestap(\varepsilon, y) = y$$
$$machinestap(0x, y) = undef(y)$$
$$machinestap(1x, y) = volgendeconfiguratie(machinestap(x, y))$$

Als x een unaire string is dan levert $machinestap(x,y)$ dus een codering van de configuratie van machine M na $\bar{x}$ stappen uitgaande van beginconfiguratie y. Als dat aantal stappen niet mogelijk is of als y geen toegelaten configuratie voorstelt dan is $machinestap(x,y)$ niet gedefinieerd.

Nu kunnen we de partiële functie $f_M:B^n \to B$ definiëren, die door M wordt berekend na input van een codering z van $(x_1, x_2, \ldots, x_m)$:

$$f_M(z) = output(machinestap(aantstap(z), input(z)))$$

met

$$aantstap(z) = \mu y \cdot alseen(unair(y), halt(machinestap(y, input(z))), nul(y))$$

Dit betekent dat f_M een partieel recursieve functie is en hiermee is de schets van het bewijs voltooid. We kunnen nog opmerken dat de methode van de onbegrensde minimalisatie maar op een plaats werd toegepast.

De stelling van Kleene biedt ons de mogelijkheid te bewijzen dat een functie Turing-berekenbaar is. Immers als we kunnen bewijzen dat een functie partieel recursief is dan volgt hieruit Turing-berekenbaarheid. Dit belangrijke resultaat is in overeenstemming met het belang dat in de informatica wordt gehecht aan recursieve structuren.

De klasse van partieel recursieve functies werd oorspronkelijk ontwikkeld als een formalisering van de klasse van effectief berekenbare functies. Uit de stelling van Kleene volgt nu dat het hier slechts gaat om een variant van de **These van Church**.

OEFENINGEN

1. Laat zien dat er aftelbaar oneindig veel berekenbare totale functies $B \to B$ bestaan. Gebruik de diagonalisatie methode om aan te tonen dat het aantal totale functies $B \to B$ niet-aftelbaar oneindig is en leid hieruit af dat het aantal niet-berekenbare totale functies $B \to B$ eveneens niet-aftelbaar oneindig is.

2. Bereken $Ack(011)$. Bewijs met inductie dat Ackermann's functie een totale functie $B \to B$ is.

3. Als g een totale functie $B^{n+1} \to B$ is en als h_1, h_2 totale functies $B^{n+1} \to B (n \geqslant 1)$ zijn dan bestaat er *precies* een functie $f: B^n \to B$ die voldoet aan

$$f(\varepsilon, x_2, \ldots, x_n) = g(x_2, \ldots, x_n)$$
$$f(0x_1, x_2, \ldots, x_n) = h_1(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n))$$
$$f(1x_1, x_2, \ldots, x_n) = h_2(x_1, x_2, \ldots, x_n, f(x_1, x_2, \ldots, x_n))$$

   Bewijs dit. Geldt deze bewering ook voor partiële functies?

4. Laat zien dat de totale functies *voet* en $top: B \to B$ die we in hoofdstuk 1 definieerden, primitief recursief zijn.

5. Definieer totale functies voor de rekenkundige bewerkingen $-$, $\times$ en $+$. Ga uit van een unaire codering en toon aan dat deze drie functies primitief recursief zijn.

6. Toon aan dat de functie $unbin:B\to B$ die is gedefinieerd door

$$unbin(x) = \begin{cases} 0 & \\ y & \text{anders, waarbij } y \text{ de binaire representatie van } \bar{x} \text{ voorstelt} \end{cases}$$

primitief recursief is.

7. Als $P_1(x_1,x_2,\ldots,x_n)$ en $P_2(x_1,x_2,\ldots,x_n)$ primitief recursief zijn dan is $(P_1 \mid P_2)(x_1,x_2,\ldots,x_n)$ dat ook. Hierbij geldt $(P_1 \mid P_2)(x_1,x_2,\ldots,x_n) = 0$ dan en slechts dan als zowel $P_1(x_1,x_2,\ldots,x_n) = 1$ als ook $P_2(x_1,x_2,\ldots,x_n) = 1$. Toon dit aan. Laat vervolgens zien dat $\sim P_1(x_1,x_2,\ldots,x_n)$ en $(P_1 \vee P_2)(x_1,x_2,\ldots,x_n)$ beide louter in termen van de koppel-operator $\mid$ kunnen worden gedefinieerd.

8. Als P een n-air primitief recursief predicaat is dan kunnen we een n-aire functie $f(n \geqslant 1)$ als volgt definiëren met behulp van *begrensde minimalisatie*. $f(x_1,x_2,\ldots,x_n)$ is het *eerste* woord y in de lexicografische ordening van B zodanig dat $P(x_1,x_2,\ldots,x_n,y) = 1$ of $y = z$. Laat zien dat de functie f noodzakelijk primitief recursief is.

9. We zeggen dat een functie $f:B\to B$ gedefinieerd is *van geval tot geval* als de volgende definitie wordt gebruikt

$$f(x) = \begin{cases} b_1 & \text{als} \quad x = a_1 \\ b_2 & \text{als} \quad x = a_2 \\ \vdots & \qquad\quad \vdots \\ b_{n-1} & \text{als} \quad x = a_{n-1} \\ b_n & \text{anders} \end{cases}$$

voor bepaalde $a_1,a_2,\ldots,a_{n-1} \in B$ en willekeurige $b_1,b_2,\ldots,b_n \in B \cup$ {ongedefinieerd}. Toon aan dat iedere unaire van geval tot geval gedefinieerde functie $B\to B$ partieel recursief is. Breid de definitie uit tot n-aire functies $B^n \to B(n > 1)$.

10. Als een functie $f:B\to B$ TM-berekenbaar is dan kan de functie worden gedefinieerd als een partieel recursieve functie met slechts een toepassing van onbegrensde minimalisatie en een primitief recur-

sief predicaat. [Hint: gebruik stelling 2.1 en het bewijs van stelling 5.7.] Leid hieruit af dat iedere primitief recursieve functie kan worden gedefinieerd met behulp van hoogstens een toepassing van onbegrensde minimalisatie en gebruik van een primitief recursief predicaat.

11. Bewijs dat de functies *volgendeconfiguratie* en *output* die werden gebruikt in het bewijs van stelling 5.7, inderdaad partieel recursief zijn.

12. Toon aan dat Ackermann's functie partieel recursief is.

6

Complexiteitstheorie

Ye gods! Annihilate but space and time

O Goden! Vernietig 't al; behoudt ruimte en tijd

Alexander Pope
The Art of Sinking in Poetry

ANALYSE VAN ALGORITMEN

We maakten in het voorgaande kennis met twee typen problemen - oplosbare problemen en problemen die, zoals het halting probleem, niet oplosbaar zijn. We weten uit ervaring dat sommige oplosbare problemen veel eenvoudiger zijn dan andere. Het sorteren van een rij van n gehele getallen bijvoorbeeld is een betrekkelijk gemakkelijk op te lossen probleem en ditzelfde geldt voor het bepalen van de minimale opspannende boom van een graaf. Het oplossen van een moeilijk probleem zal in het algemeen veel computertijd in beslag nemen en vaak is er ook veel geheugenruimte voor nodig. Voordat we echter mogen beweren dat een probleem moeilijk is, zullen we er eerst zeker van moeten zijn dat het probleem niet alleen maar moeilijk lijkt omdat we een verkeerde oplossingsmethode toepassen. Met andere woorden we willen er zeker van zijn dat zelfs het best mogelijke algoritme voor het oplossen van het probleem veel rekentijd nodig heeft en mogelijk ook op veel geheugenruimte beslag legt.

In dit hoofdstuk zullen we deze overwegingen formaliseren en we zullen vragen beantwoorden zoals: Wanneer is een probleem gemakkelijk? Wanneer is het ene algoritme beter dan het andere? Wanneer is er voor een oplossing veel geheugenruimte nodig? De tak van de berekenbaarheidstheorie die zich met het beantwoorden van dit soort vragen bezig houdt wordt *complexiteitstheorie* genoemd. Veel resultaten die we hier beschrijven dateren van na 1970. De complexiteitstheorie is nog steeds een belangrijk onderwerp van onderzoek en de resultaten hebben

belangrijke repercussies voor toepassingen binnen de informatica. De belangrijkste ideeën zullen we hier behandelen en we zullen daarbij zeker ingaan op de theorie van de NP-complete problemen.

De tijd die nodig is om een bepaald probleem op te lossen hangt bijna altijd af van de 'omvang' van het speciale geval dat moet worden opgelost en ook van het algoritme dat voor het oplossen gebruikt wordt. De probleemomvang wordt meestal op een nogal informele manier gemeten; er wordt een verband gelegd met de hoeveelheid input die nodig is om het speciale geval te beschrijven. Laten we bijvoorbeeld eens het probleem bekijken van het vermenigvuldigen van twee vierkante matrices met geheeltallige elementen. Als het in een speciaal geval gaat om het vermenigvuldigen van twee $n \times n$ matrices dan kunnen we n als maat voor de probleemomvang gebruiken, maar we zouden natuurlijk ook n^2 of $2n^2$ kunnen gebruiken. Als we de standaardmethode voor matrixvermenigvuldiging gebruiken dan wordt het element (i,j) berekend uit het scalaire produkt van de i-de rij uit de eerste matrix en de j-de kolom van de tweede matrix. Het berekenen van dit produkt, (als we ervan uitgaan dat er geen overflow optreedt omdat de getalomvang de capaciteit van de binnen de computer voorstelbare gehele getallen overschrijdt), vergt $2n$ lees-operaties, n vermenigvuldigingen, $n-1$ optellingen en een schrijf-operatie. De totale matrixvermenigvuldiging vergt dus $2n^3$ lees-operaties, n^3 vermenigvuldigingen, $n^2(n-1)$ optellingen en n^2 schrijf-operaties. Hiermee hebben we een maat voor de hoeveelheid werk (en dus ook voor de tijd) die nodig is om twee matrices met behulp van dit algoritme te vermenigvuldigen.

Bij de analyse van algoritmen (meestal in de een of andere hogere programmeertaal geschreven) door informatici worden inderdaad de benodigde elementaire operaties geteld. Elementaire operaties zijn de rekenkundige en logische operaties, verder bestandsbenaderingen, het inlezen van waarden van variabelen en het wegschrijven van waarden naar variabelen. De *tijdcomplexiteit* van een algoritme A, $tijd_A(n)$ wordt nu uitgedrukt als een functie van de probleemomvang n. De waarde van $tijd_A(n)$ wordt gewoonlijk gedefinieerd als het grootste aantal primitieve operaties dat door het algoritme, gegeven een inputomvang n, zou kunnen moeten worden uitgevoerd. Dat wil zeggen dat $tijd_A(n)$ een 'slechtste geval'-maat is. De analyse wordt vaak ter bepaling van de gemiddelde of verwachte complexiteit uitgevoerd, maar in deze inlei-

ding zullen we ons alleen met de meestal eenvoudiger te analyseren complexiteit in het slechtste geval bezighouden.

Sommige primitieve operaties nemen meer tijd in beslag dan andere en de werkelijk benodigde tijd is sterk machine-afhankelijk. Ook is er bij het praktisch uitvoeren van algoritmen altijd sprake van extra benodigde tijd voor adresberekeningen, controle van arraygrenzen, overflow-afhandeling, type checking enzovoorts. Al deze zaken zullen wij niet bij onze analyse betrekken en onze definitie van tijdcomplexiteit is dus vrij ruw en niet direct in de praktijk toepasbaar. Ook op een ander punt kan men nog kritiek hebben op onze definitie van tijdcomplexiteit: bij kleine problemen is de rekentijd vaak helemaal niet afhankelijk van de probleemomvang, maar louter van de tijd die nodig is om het programma op te starten. Meestal worden resultaten over de tijdcomplexiteit van algoritmen daarom weergegeven met behulp van de zogenaamde 0-notatie (orde-notatie).

Stel f, g zijn functies $Z^+ \rightarrow Z^+$. We zeggen dan $f(n)$ is $0(g(n))$, lees: $f(n)$ is van de orde $g(n)$ desd als er positieve constanten c en N bestaan zodanig dat $f(n) \leqslant cg(n)$ voor alle $n \geqslant N$.

Zo is $3n^2 + 4n - 3$, $0(n^2)$ omdat $3n^2 + 4n - 3 \leqslant 4n^2$ voor alle $n \geqslant 3$. Sterker nog: iedere $a_2n^2 + a_1n + a_0$ is $0(n^2)$ omdat $a_2n^2 + a_1n + a_0 \leqslant (|a_2| + |a_1| + |a_0|)n^2$ voor alle $n \geqslant 1$. Uit de definitie volgt niet automatisch dat een $0(n^2)$ functie een term in n^2 moet bevatten. Iedere $0(n)$ functie is ook $0(n^2)$ en ditzelfde geldt voor iedere $0(n \log n)$ functie. Het omgekeerde van deze bewering is echter niet waar.

De matrixvermenigvuldigingsmethode die we eerder in dit hoofdstuk bespraken is een $0(n^2)$ algoritme. Verdere voorbeelden kan de geïnteresseerde lezer vinden in *The Design and Analysis of Computer Algorithms* (Aho, Hopcroft & Ullman, 1974) of in *Fundamentals of Computer Algorithms* (Horowitz & Sahni, 1978).

Veronderstel dat de algoritmen A en B een bepaald probleem Π oplossen met tijdcomplexiteit van respectievelijk $tijd_A$ en $tijd_B$ dan noemen we algoritme A *beter dan* algoritme B voor het oplossen van probleem Π als

(1) $tijd_A$ is $0(tijd_A)$, maar
(2) $tijd_B$ is niet $0(tijd_A)$.

Er bestaan betere algoritmen voor matrixvermenigvulding dan het algoritme dat we hier beschreven. Zie bijvoorbeeld het algoritme van Stras-

sen (oefening 6.3). Dit is $0(n^{2.81})$, maar het is bekend dat ook dit nog niet het beste algoritme is.

Bovenstaande definitie van 'beter dan' is nogal misleidend. Veronderstel dat we twee algoritmen A en B hebben met

$$tijd_A(n) = \begin{cases} 1.000.000 & \text{voor } n \leqslant 1.000.000 \\ n & \text{voor } n > 1.000.000 \end{cases}$$

en

$$tijd_B(n) = \begin{cases} n & \text{voor } n \leqslant 1.000.000 \\ n^2 & \text{voor } n > 1.000.000 \end{cases}$$

dan is $tijd_A(n)$, $0(tijd_B(n))$ omdat $tijd_A(n) \leqslant tijd_B(n)$ voor alle $n \geqslant 1.000.000$ maar $tijd_A(n)$ is niet $0(tijd_A(n))$. Volgens onze definitie is algoritme A dus beter dan algoritme B, maar in werkelijkheid komt het misschien nooit voor dat we met een probleemomvang van meer dan 1.000.000 werken. In de praktijk zullen we dus algoritme B boven algoritme A prefereren. Gelukkig lijkt het erop dat we in de praktijk dergelijke situaties vrijwel nooit tegen zullen komen en we kunnen dus de 0-notatie en de 'beter dan' definitie wel degelijk gebruiken als goede gereedschappen bij het analyseren van algoritmen. De constanten binnen tijdcomplexiteitsfuncties zijn gewoonlijk betrekkelijk klein en het gedrag van de gebruikte functies is niet pathologisch.

Behalve de 0-notaties worden ook de Ω- en de θ-notatie gebruikt bij de analyse van algoritmen. Als f en g functies $Z^+ \to Z^+$ zijn dan noemen we $f(n)$ $\Omega(g(n))$ desd als er positieve constanten c en N bestaan zodanig dat $f(n) \geqslant g(n)$ voor alle $n \geqslant N$. Zo is bijvoorbeeld $3n^2 + 4n - 3$ $\Omega(n^2)$ maar ook $\Omega(n)$ en $\Omega(n \log n)$. Als we weten dat ieder algoritme dat een probleem Π oplost een tijdcomplexiteitsfunctie moet hebben dat $\Omega(g(n))$ is voor een of andere functie $g(n)$ dan weten we ook dat ieder $0(g(n))$ algoritme dat Π oplost in bepaalde zin optimaal is. Zolang we echter zo'n algoritme niet hebben gevonden hoeven we er niet zeker van te zijn dat een dergelijk algoritme ook bestaat.

Als f, g functies $Z^+ \to Z^+$ zijn dan heet $f(n)$ $\theta(g(n))$ desd als $f(n)$ zowel $0(g(n))$ als $\Omega(g(n))$ is. Als f en g polynomen zijn en $f(n)$ is $\theta(g(n))$ dan moeten dus f en g van dezelfde orde zijn, maar de functies hoeven niet identiek te zijn omdat er sprake kan zijn van verschillende waarden van constanten.

We noemen een algoritme van *polynomiale tijd* desd als zijn tijdcomplexiteit $O(n^k)$ is voor een $k \geqslant 0$. In de praktijk geldt dat als een probleem oplosbaar is met behulp van een algoritme van polynomiale tijd, we meestal een algoritme kunnen vinden met een tijdcomplexiteitsfunctie $O(n^k)$ met een vrij kleine k, (meestal kleiner dan 5). Specialisten in het ontwerpen en analyseren van algoritmen besteden meestal veel tijd aan het zoeken naar efficiënte algoritmen voor specifieke problemen. Er bestaat echter een hele klasse van problemen waarvoor het tot nu toe niet gelukt is om algoritmen van polynomiale tijd te vinden. Een algoritme met een tijdcomplexiteitsfunctie $\theta(c^n)$ voor een of andere $c > 1$ heet een algoritme van *exponentiële tijd*. Dergelijke algoritmen kunnen zich in het slechtste geval nogal desastreus gedragen. Stel bijvoorbeeld dat een algoritme A dat een probleem Π oplost voor een probleemomvang n tijd 2^n nodig heeft. Als Π ook door een algoritme B kan worden opgelost in tijd n^5 dan kan men uitrekenen dat voor een probleemomvang $n = 60$, B het probleem in een minuut oplost, terwijl A er 28000 jaar voor nodig heeft!

Bij een redelijke probleemomvang kunnen niet-polynomiale algoritmen behoorlijk tijdrovend zijn, maar we moeten daarbij niet vergeten dat we het steeds hebben over het slechtste geval. Het is best mogelijk dat alleen in enkele uitzonderingsgevallen werkelijk zeer veel rekentijd nodig is terwijl het algoritme in de praktijk tot volle tevredenheid bruikbaar is. Zo wordt voor het oplossen van lineaire programmeringsproblemen meestal het simplexalgoritme gebruikt dat in het slechtste geval van exponentiële tijd is. Toch blijkt het in de praktijk erg goed te werken en wordt het verkozen boven de zogenaamde ellipsoïdemethode die wel van polynomiale tijd is.

Als we voor een probleem Π alleen een algoritme hebben dat niet van polynomiale tijd is dan zullen we meestal proberen om een wel polynomiaal algoritme te vinden. Maar wat als dat niet lukt? Dat kan aan ons liggen, maar het kan ook aan het probleem Π liggen - misschien bestaat er wel helemaal geen polynomiaal algoritme. We noemen een probleem *onhandelbaar* (Engels: intractable) als er geen polynomiaal algoritme voor bestaat, maar als het wel oplosbaar is. Alles wijst erop dat er klassen van dergelijke problemen bestaan en in het vervolg van dit hoofdstuk zullen we een theorie ontwikkelen die deze bewering ondersteunt.

Voordat we nader ingaan op deze theorie geven we een voorbeeld van een probleem waarvan men algemeen denkt dat het onhandelbaar is omdat nog niemand een polynomiaal algoritme heeft gevonden om het op te lossen. Uit de theorie die we zullen ontwikkelen zal blijken dat het ook niet waarschijnlijk is dat dit ooit zal lukken! Het probleem heet het *handelsreizigersprobleem* (Engels: Traveling Salesman Problem of TSP). Gegeven zijn n steden, $c_1, c_2, \ldots, c_n$ en een handelsreiziger bevindt zich in stad c_1. Tussen elk paar steden c_i, c_j is een afstand d_{ij} gegeven. De handelsreiziger vertrekt vanuit c_1 en wil elk der steden $c_2, \ldots, c_n$ *precies een keer* bezoeken en vervolgens naar huis terugkeren. Het kan hem niet schelen in welke volgorde hij de steden bezoekt, maar wel wil hij totaal een zo klein mogelijke afstand afleggen. Het op te lossen probleem is om deze rondreis met minimale kosten te bepalen. Als we er maar genoeg tijd aan besteden dan is het oplossen van het handelsreizigersprobleem niet zo moeilijk: bekijk eenvoudig alle $(n-1)!$ mogelijke permutaties van de steden $c_2, \ldots, c_n$ en bereken de bijbehorende kosten. Selecteer vervolgens de goedkoopste rondreis.

Net als in hoofdstuk 3 zullen we ons ook hier weer hoofdzakelijk bezighouden met beslissingsproblemen en niet met optimaliseringsproblemen zoals het handelsreizigersprobleem. We zagen echter ook al eerder dat uit elk optimaliseringsprobleem een bijbehorend beslissingsprobleem kan worden afgeleid. Bij het handelsreizigersprobleem hoort het volgende beslissingsprobleem:

TSP

Gegeven: Een verzameling $C = \{c_1, c_2, \ldots, c_n\}$ van n steden en bij ieder paar $c_i, c_j \in C$ een afstand $d_{ij} \in Z^+$ en een grens $b \in Z^+$.

Gevraagd: Kan een handelsreiziger vanuit stad c_1 elk der steden in $\{c_2, \ldots, c_n\}$ precies een keer bezoeken en vervolgens naar huis terugkeren, waarbij de totale afgelegde afstand $\leqslant b$ is?

Algoritmen worden traditioneel als deterministisch beschouwd, dat wil zeggen bij iedere stap in het algoritme ligt de volgende stap eenduidig vast. Gewoonlijk laten we de constructie *of* dan ook niet toe in onze algoritmebeschrijvingen. Zouden we echter over mogelijkheden tot paral-

le verwerking beschikken dan zou die constructie wel kunnen worden gebruikt. Een opdracht zoals

doe A of doe B

zou parallel kunnen worden uitgevoerd. Een processor zou A kunnen uitvoeren terwijl de tweede B behandelt. De mogelijkheid tot parallelle verwerking geeft dus aanleiding tot niet-deterministische algoritmen. Zouden we beschikken over mogelijkheden tot onbegrensde parallelle verwerking (zover zal het echter nooit komen) dan zouden we TSP in polynomiale tijd kunnen oplossen. Elk van de processoren zou dan een van de $(n-1)!$ mogelijke permutaties nemen en de bijbehorende afstand berekenen. Vervolgens zouden we het resultaat bekijken en nagaan of er ook een afstand $\leqslant b$ bij was. Als we maar een processor hebben dan zouden we eigenlijk moeten kunnen 'raden' welke van de $(n-1)!$ permutaties moet worden berekend. Als we goeg raden en het antwoord is 'ja' dan hebben we het probleem opgelost. We zullen in het vervolg zien dat het idee van een niet-deterministisch programma een belangrijke toepassing heeft in de theorie die we nu gaan beschrijven.

DE KLASSEN P EN NP

Wat we hiervoor behandelden gaan we nu formaliseren en als rekenmodel gebruiken we daarbij de Turingmachine. Net als in hoofdstuk 3 beperken we ons tot beslissingsproblemen met ja/nee antwoorden.

Beschouw een beslissingsprobleem Π en een codering e van instanties I van Π in strings van T^*. Een algoritme dat Π oplost is dan een TM, $M = (Q, \Sigma, T, P, q_0, F)$ die kan bepalen of een input $e(I)$ al dan niet in de taal $L_{\Pi, e} = \{e(I) | I \in Y_\Pi\}$ voorkomt.

We beperken onze coderingen e tot geschikt compacte coderingen. Hiermee bedoelen we dat (i) geen codering van een instantie I bevat overbodige opvulsymbolen, (ii) getallen of andere symbolen in I worden in binaire notatie gecodeerd (of octaal, of decimaal enzovoorts, maar *niet* unair) en (iii) $e(I)$ kan eenduidig en effectief naar I worden gedecodeerd.

Stel we hebben een deterministische TM, $M = (Q, \Sigma, T, P, q_0, F)$ en

we voeren in de string $x \in T^*$. We definiëren nu de *tijdcomplexiteitsfunctie van een deterministische TM, M,* $tijd_M : Z^+ \to Z^+$ als

$$tijd_M(n) = \max\{m \mid \text{er bestaat een } x \in T^n \text{ zodanig dat berekening van } M \text{ na input } x \text{ lengte } m \text{ heeft}\}.$$

De lengte van een berekening van een deterministische TM na een gegeven input is, zoals we eerder zagen, gedefinieerd als het aantal configuraties dat wordt doorlopen voordat de machine stopt. De lengte van de berekening komt dus exact overeen met de tijd die de TM voor de berekening nodig heeft. Als M een algoritme is en dus na elke input na eindige tijd stopt dan is $tijd_M(n)$ gedefinieerd voor alle n. Als dit niet het geval is dan kan M in een oneindige lus geraken na een of andere input x en $tijd_M(|x|)$ is dan oneindig.

Als $tijd_M(n)$ $0(n^k)$ is voor een $k \geqslant 0$ dan is M van *polynomiale tijd*. Een deterministische TM van polynomiale tijd is noodzakelijk een algoritme omdat de machine na iedere input uiteindelijk zal stoppen. We definiëren nu een klasse talen P door

$$P = \{L \mid \text{er bestaat een deterministische TM van polynomiale tijd die } L \text{ accepteert}\}.$$

Elke taal in P is dus een recursieve taal (stelling 4.1). Hoewel P formeel is gedefinieerd als een klasse talen, kunnen we P ook beschouwen als een klasse beslissingsproblemen. We schrijven $\Pi \in P$ desd als er een codering e van Π bestaat zodanig dat $L_{\Pi,e} \in P$. P staat dus voor alle problemen die in polynomiale tijd kunnen worden opgelost door deterministische algoritmen.

Laten we nu eens uitgaan van een nondeterministische Turingmachine $M = (Q, \Sigma, T, P, q\ , F)$. We zeggen dat M het beslissingsprobleem Π onder codering e oplost desd als $T(M) = L_{\Pi,e}$. Dit betekent dus dat er bij elk probleemvoorkomen $I \in Y_\Pi$, na input $e(I)$ in M, *een of andere* rij van mogelijke stappen bestaat die leidt tot het met succes stoppen van M.

Stel M is een nondeterministische Turingmachine (NDTM) die de string $x \in T^*$ accepteert. Er bestaat dan minstens één rij van acties (rekenstappen) waarbij de hele string x en mogelijk meer door M wordt geaccepteerd. De tijd die M nodig heeft om x te accepteren is gedefini-

eerd als de lengte van de kortste van die mogelijke rijen van rekenstappen. De tijdcomplexiteitsfunctie $tijd_M$ voor een NDTM, M is alleen afhankelijk van het aantal stappen nodig om de input te accepteren. Als inputs van lengte n niet door M worden geaccepteerd dan spreken we af dat $tijd_M(n) = 1$. De *tijdcomplexiteitsfunctie* voor een NDTM, M wordt dus gedefinieerd als

$$tijd_M(n) = \begin{cases} 1 & \text{als } T(M) \cap T^n = \phi, \\ \min\{m \mid \text{er bestaat een } x \in T^n \\ \quad \text{zodanig dat de tijd nodig} \\ \quad \text{voor het accepteren van } x \\ \quad \text{door } M \text{ gelijk is aan } m\} & \text{anders} \end{cases}$$

M heet een NDTM van polynomiale tijd desd als $tijd_M(m)$ $0(n^k)$ is voor een of andere $k \geqslant 0$.

We kunnen nu de klasse NP definiëren - dit is de klasse van talen (en dus van beslissingsproblemen) die door NDTM's in polynomiale tijd worden geaccepteerd. Namelijk

$$NP = \{L \mid \text{er bestaat een NDTM van polynomiale tijd die } L \text{ accepteert}\}$$

Elke taal in NP is dus recursief aftelbaar. Net als P komt ook NP behalve met een klasse van talen overeen met een klasse van beslissingsproblemen. Er geldt $\Pi \in NP$ desd als er een codering e van Π bestaat zodanig dat $L_{\Pi,e} \in NP$. NP vertegenwoordigt de problemen die in polynomiale tijd oplosbaar zijn door een nondeterministisch programma.

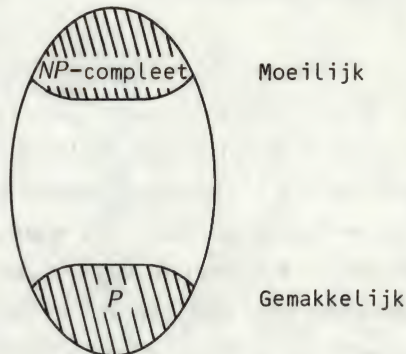
Uit het voorgaande en op grond van wat we behandelden in hoofdstuk 4 zal het duidelijk zijn dat $P \subseteq NP$. Ook lijkt het zeer waarschijnlijk dat $P \neq NP$ - alle geleerden zijn het er eigenlijk over eens dat deze bewering waar is, maar tot nu toe heeft niemand het kunnen bewijzen. Dit is wel zeer betreurenswaardig omdat de theorie die we hierna zullen ontwikkelen volledig op deze veronderstelling gebaseerd is. Mocht ooit worden bewezen dat wel geldt dat $P = NP$ dan gelieve de lezer de volgende pagina's uit dit boek te scheuren!

Een beslissingsprobleem dat zich in de klasse NP bevindt is, het handelsreizigersprobleem TSP. De kern van de oplossingsmethode

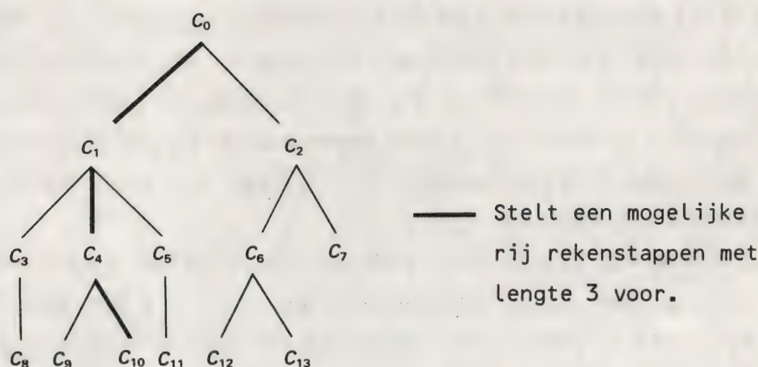
bestaat uit het uitschrijven door de NDTM van een gissing naar de oplossing in de vorm van een rondreis behorend bij een specifiek probleem en uit het vervolgens berekenen van de bijbehorende kosten. Zijn die kosten kleiner of gelijk b dan is het antwoord op de gestelde vraag 'ja'. Als de NDTM de juiste stappen uitvoert (en dus goed raadt) dan lost de machine het probleem op.

NP bevat veel problemen zoals TSP die niet in P lijken te zitten. Sterker nog: zouden we ooit kunnen aantonen dat $P \neq NP$ dan volgt hieruit dat TSP $\notin P$. TSP is een voorbeeld van een NP-compleet probleem. Dit is een klasse van problemen in NP die in bepaalde zin de moeilijkste problemen in NP vertegenwoordigt. In figuur 6.1 is de situatie schematisch weergegeven.

Voordat we het begrip NP-compleet formeel definiëren, zullen we eerst de klasse NP nog wat verder onderzoeken. Zij $M = (Q, \Sigma, T, P, q_0, F)$ een NDTM die een taal $L \subseteq T^*$ herkent. Als M zich in een bepaald stadium van de berekening in configuratie $C = (q, i, \alpha, a, \beta)$ bevindt dan zijn er misschien, omdat $\#P(q,a) > 1$ kan zijn, verschillende toegelaten stappen mogelijk. Een configuratie die uit C volgt door het uitvoeren van een toegelaten stap, noemen we een *kind* van C. Grafisch geven we dat weer met behulp van een *configuratieboom*. In de wortel van de boom plaatsen we C_0 de beginconfiguratie. Deze knoop wordt vervolgens verbonden met alle kinderen van C_0 en deze kinderen weer met hun kinderen. Zie figuur 6.2. Heeft een configuratie in de boom geen kinderen dan wordt deze een blaadje in de boom. Een dergelijke configuratie wordt een *acceptatieconfiguratie*



Figuur 6.1 De klasse NP



Figuur 6.2 Een voorbeeld van een configuratieboom

genoemd desd als het om een situatie gaat waarin M zich in een eindtoestand bevindt. Ieder pad vanuit de wortel via een aantal knopen naar een acceptatieconfiguratie stelt een rij rekenstappen voor die de oorspronkelijke input accepteert. Als we een nieuwe toestand $q_y \notin Q$ introduceren met $P(q,a) = \{(q_y,a,0)\}$ voor alle $q \in F$ en als we vervolgens F opnieuw definiëren als $\{q_y\}$ dan bevindt de NDTM, M zich in een acceptatieconfiguratie desd als M zich in toestand q_y bevindt. Een dergelijke toestand noemen we een *acceptatietoestand*. Hiermee bewezen we tegelijkertijd de volgende stelling.

Stelling 6.1

Als L wordt geaccepteerd door een NDTM, M dan bestaat er ook een NDTM, M' met precies één eindtoestand (de acceptatietoestand) die L accepteert.

Merk op als we x in M' invoeren en M' komt in de acceptatietoestand dat hieruit volgt dat $x \in L$. Het omgekeerde geldt echter niet, dat wil zeggen als M' in een lus komt of in een *weigertoestand* dan mogen we niet concluderen dat $x \notin L$. Het is immers ook mogelijk dat we alleen maar op het verkeerde pad zijn geraakt in de configuratieboom.

Laten we nu eens een string $x \in T^*$ bekijken. Als x wordt ingevoerd in de NDTM, M dan is de beginconfiguratie

$C_0(x) = (q_0, 1, \varepsilon, als\ x = \varepsilon\ dan \wedge anders\ kop(x), staart(x))$. De string x wordt door M geaccepteerd desd als er een pad bestaat in de configuratieboom vanuit de wortel $C_0(x)$ naar een acceptatieconfiguratie. De lengte van een dergelijk pad bepaalt de tijd die voor de berekening nodig is. M is dus een NDTM van polynomiale tijd die L accepteert desd als er een $k \geqslant 0$ bestaat zodanig dat voor iedere $x \in L$ het kortste pad in de boom vanuit $C_0(x)$ tot aan een acceptatieconfiguratie lengte $l_M(x)$ heeft, waarbij $l_M(x)$ $0(|x|^k)$ is.

Het maximum aantal kinderen van een configuratie is $graad(x) = \max\{\#P(q,a) \mid q \in Q, a \in \Sigma\}$. Dit is de *graad* van de NDTM, M. De eerste l niveaus van een configuratieboom van een NDTM met graad $m > 1$ hebben noodzakelijk $\leqslant \sum_{k=0}^{l} m^k = (m^{l+1} - 1)/(m - 1)$ knooppunten.

Stelling 6.2

Als L wordt geaccepteerd door een NDTM, M dan bestaat er ook een NDTM, M_2 met graad $\leqslant 2$ die L accepteert. Als verder M, L in polynomiale tijd accepteert dan doet M_2 dat ook.

Bewijs. Zij $M = M_m = (Q, \Sigma, T, P, q_0, F)$ een NDTM met graad $m > 2$. We laten nu eerst zien hoe we een NDTM $M_{m-1} = (Q', \Sigma, T, P', q_0, F)$ van graad $m - 1$ kunnen construeren die L ook accepteert.

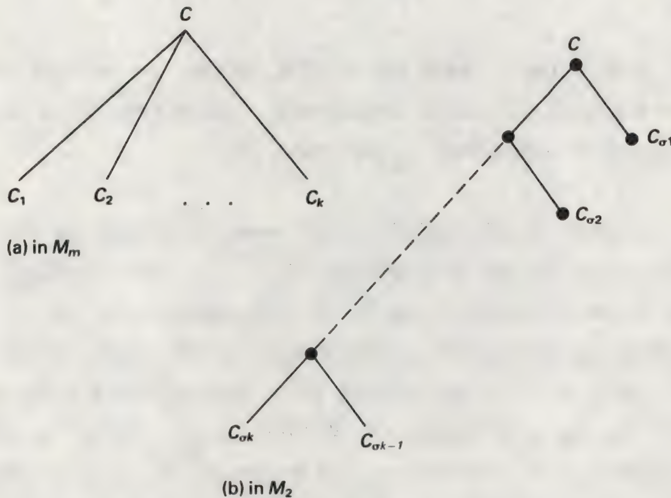
Veronderstel dat er maar r verschillende paren $(q,a) \in Q \times \Sigma$ bestaan met $\#P(q,a) = m$. We voeren nu r nieuwe toestanden in $q_1', q_2', \ldots, q_r'$ en we definiëren $Q' = Q \cup \{q_1', q_2', \ldots, q_r'\}$. We definiëren verder P' gelijk aan P behoudens deze r paren. Als (q,a) het i-de paar is in een of andere aftelling en als $P(q,a) = \{(q_1, b_1, X_1), (q_2, q_2, X_2), \ldots, (q_m, b_m, X_m)\}$ dan definiëren we

$$P'(q,a) = \{(q_1, b_1, X_1), (q_1', a, 0)\}$$

en $$P'(q_1', a) = \{(q_2, b_2, X_2), \ldots, (q_m, b_m, X_m)\}.$$

Duidelijk is dat $T(M_m) = T(M_{m-1})$ en M_{m-1} is van graad $m - 1$. Dit proces kunnen we voortzetten en we krijgen zo een rij van NDTM's, $M_m, M_{m-1}, \ldots, M_2$ die alle L accepteren. Hiermee is het eerste gedeelte van de stelling bewezen.

Stel nu dat $M = M_m$ de string x accepteert. Dit betekent dat er in de configuratieboom van M met wortel $C_0(x)$ een pad bestaat met lengte $l_M(x)$. Veronderstel dat er een knoop op dit pad zit met graad k $(1 < k \leqslant m)$ en dat die knoop bij configuratie C behoort. De kinderen van C zijn bijvoorbeeld $C_1, C_2, \ldots, C_k$. Op grond van de constructie van M_2 kan de overgang van C naar een van zijn kinderen worden weergegeven zoals in figuur 6.3(b) voor de een of andere permutatie σ van $1, 2, \ldots, k$. Het pad in de configuratieboom bij M_2 dat een overgang aangeeft van C naar een van de kinderen $C_1, C_2, \ldots, C_k$ heeft een lengte van hoogstens $k - 1 \leqslant m - 1$. Er bestaat dus een pad vanuit de wortel $C_0(x)$ naar een acceptatieknoop in de configuratieboom van M_2 met lengte $l_{M_2}(x) \leqslant (m-1) l_M(x)$. Als nu $l_M(x)$ $0(|x|^k)$ is dan geldt dit ook voor $(m-1) l_M(x)$ en dus ook voor $l_{M_2}(x)$. Hiermee is ook het tweede deel van de stelling bewezen.



Figuur 6.3

DE KLASSE NP-COMPLEET

Veronderstel dat we twee problemen hebben Π_1 en Π_2 en verder een algoritme A dat Π_2 oplost. Als elke instantie I van Π_1 gemakkelijk zou kunnen worden getransformeerd naar een voorkomen van Π_2 dan konden

we A ook gebruiken om Π_1 op te lossen. Als we bijvoorbeeld een polynomiaal algoritme hadden voor TSP dan zou dit ook het volgende probleem oplossen.

Hamiltoncircuit (HC)
Gegeven: Een graaf G.
Gevraagd: Bevat G een Hamiltoncircuit, dat wil zeggen een cyclisch pad dat alle knooppunten onderling verbindt?

We zullen een functie f specificeren die elk HC-probleem in een bijbehorend TSP-probleem vertaalt. De functie f kan vrij eenvoudig worden gedefinieerd. Veronderstel dat een bepaald HC-probleem is gespecificeerd door een graaf G bestaande uit een verzameling knopen (Engels: vertices) V en een verzameling kanten of verbindingen (Engels: edges) E. Het bijbehorende TSP-probleem heeft nu $C = V$ met voor elk paar steden $v_i, v_j \in C$, $d_{ij} = 1$ als v_i en v_j onderling verbonden zijn in G en $d_{ij} = 2$ als zij niet verbonden zijn. De grens b krijgt de waarde $n = |V|$ en duidelijk is dat $I \in Y_{HC}$ desd als $f(I) \in Y_{TSP}$.

Een dergelijke functie f is een voorbeeld van een reductie; een begrip dat we ook al in hoofdstuk 3 tegenkwamen. Hier zijn we geïnteresseerd in polynomiale tijd en de reducties willen we dan ook beperken tot in polynomiale tijd uitvoerbare reducties. We zeggen dat een probleem Π_1 *polynomiaal reduceert* of *transformeert naar* Π_2, notatie $\Pi_1 \propto \Pi_2$, desd als er een functie $f: D_{\Pi_1} \to D_{\Pi_2}$ bestaat met

(1) voor iedere $I \in D_{\Pi_1}$, $I \in Y_{\Pi_2}$ desd als $f(I) \in Y_{\Pi_2}$ en

(2) f kan worden berekend in polynomiale tijd.

In een formelere taaltheoretische context zeggen we dat $L_1 \subseteq T_1^*$ polynomiaal transformeert naar $L_2 \subseteq T_2^*$, notatie $L_1 \propto L_2$, desd als er een functie $f: T_1^* \to T_2^*$ bestaat zodanig dat

(1) voor alle $x \in T_1^*$, $x \in L_1$ desd als $f(x) \in L_2$ en

(2) er bestaat een deterministische TM die f in polynomiale tijd berekent.

Dus geldt $\Pi_1 \propto \Pi_2$ desd als er (geschikte en efficiënte) coderingen e_1 en e_2 bestaan zodat $L_{\Pi_1, e_1} \propto L_{\Pi_2, e_2}$.

Stelling 6.3

$\propto$ is transitief.

Bewijs. We gaan bij het bewijs uit van het taalmodel. Veronderstel dat $L_1 \propto L_2$ en $L_2 \propto L_3$. We moeten dan aantonen dat ook $L_1 \propto L_3$. Als $L_1 \subseteq T_1^*$ en $L_2 \subseteq T_2^*$ dan bestaat er een functie $f: T_1^* \rightarrow T_2^*$ die behoort bij de transformatie $L_1 \propto L_2$. Stel f wordt in polynomiale tijd berekend door een TM, M_f. Verder is er ook een functie $g: T_2^* \rightarrow T_3^*$ die in polynomiale tijd wordt berekend door een TM, M_g en die hoort bij de transformatie $L_2 \propto L_3$ met $L_2 \subseteq T_2^*$ en $L_3 \subseteq T_3^*$. We construeren vervolgens de TM, M die de acties van M_f simuleert, gevolgd door de acties van M_g. Deze deterministische Turingmachine berekent de functie $g_0 f$ in polynomiale tijd. Omdat $x \in L_1$ desd als $f(x) \in L_2$ desd als $g(f(x)) \in L_3$, definieert $g_0 f$ de transformatie $L_1 \propto L_3$.

Stelling 6.4

Als $L_1 \propto L_2$ en $L_2 \in P$ dan is $L_1 \in P$.

Bewijs. Veronderstel $L_1 \subseteq T_1^*$ en $L_2 \subseteq T_2^*$ en $f: T_1^* \rightarrow T_2^*$ is de functie die behoort bij de transformatie $L_1 \propto L_2$. Stel verder dat M_f een deterministische TM is die f in polynomiale tijd berekent. Omdat $L_2 \in P$ bestaat er een TM, M_2 van polynomiale tijd die L_2 accepteert. Als M de acties van M_f simuleert, gevolgd door de acties van M_2 dan hebben we een deterministische TM van polynomiale tijd verkregen die L_1 accepteert. Dus geldt $L_1 \in P$.

We definiëren twee talen L_1 en L_2 (en evenzo twee problemen Π_1 Π_2) als *polynomiaal equivalent* als $L_1 \propto L_2$ en $L_2 \propto L_1$ (evenzo $\Pi_1 \propto \Pi_2$ en $\Pi_2 \propto \Pi_1$). We weten dat $\propto$ transitief is op grond van stelling 6.3; met de eenheidsfunctie kan worden aangetoond dat $\propto$ reflexief is en dus is polynomiale equivalentie per definitie symmetrisch. Dit laatste betekent weer dat polynomiale equivalentie een equivalentierelatie is. P is een van de equivalentieklassen, (zie oefening 6.4) - de klasse van

de 'gemakkelijke' talen (problemen) in NP. De 'moeilijke' talen (problemen) in NP zijn de NP-complete problemen.

Een taal L is *NP-compleet* dan en slechts dan als
(1) $L \in NP$, en
(2) voor alle $L' \in NP$ geldt $L' \propto L$.

We noemen een probleem Π NP-compleet desd als $L_{\Pi,e}$ NP-compleet is voor een geschikte beknopte codering e. Informeel is Π NP-compleet desd als $\Pi \in NP$ en als voor alle $\Pi' \in NP$ geldt dat $\Pi' \propto \Pi$.

De klasse van NP-complete talen zullen we in het vervolg NPC noemen.

Stelling 6.5

(1) NPC is een equivalentieklasse onder polynomiale equivalentie.
(2) Als $NPC \cap P \neq \phi$ dan is $NP = P$.

Bewijs.
(1) Stel dat $L_1, L_2 \in NPC$ dan geldt per definitie $L_1, L_2 \in NP$. Omdat L_1 NP-compleet is geldt $L_1 \propto L_2$ en omdat L_2 NP-compleet is geldt $L_1 \propto L_2$. Dus zijn L_1 en L_2 polynomiaal equivalent.
(2) Stel $L \in NPC \cap P$. Omdat $L \in NPC$ geldt voor elke $L' \in NP$ dat $L' \propto L$. Maar ook geldt $L \in P$ en dus volgt uit stelling 6.4 dat alle $L' \in P$. In dat geval is dus $NP = P$.

In werkelijkheid wijst alles erop dat $NP \neq P$ en dus zullen we wel geen oplossingen met algoritmen van polynomiale tijd kunnen vinden voor NP-complete problemen. Trouwens als er een NP-compleet probleem was waarvoor zo'n algoritme werd gevonden dan was er een polynomiaal algoritme voor alle problemen in NP. Het eerst bekende NP-complete probleem was het voldoeningsprobleem, (Engels: satisfiability problem) voor een verzameling clausen.

Gegeven is een verzameling Boole'se variabelen, $U = \{u_1, u_2, \ldots, u_n\}$. Een *literaal* uit U is gedefinieerd als ofwel een van deze variabelen, ofwel de ontkenning van een $u \in U$, (meestal geschreven als $\bar{u}$). Een *claus* over U is een deelverzameling van literalen uit U. Zo is

$\{u_2, u_4, u_5\}$ een claus over U die bestaat uit drie literalen. We zeggen dat aan een claus is *voldaan* door middel van een toekenning van waarden (waar of onwaar) aan de Boole'se variabelen als minstens een van de literalen uit de claus de waarde 'waar' heeft. Dus aan $\{u_2, \bar{u}_4, u_5\}$ is voldaan door een waardetoekenning dat u_2 waar is, of $\bar{u}_4$ onwaar, of u_5 waar. Laat C nu een verzameling van clausen over U voorstellen. We zeggen dan dat aan C *voldaan* kan worden dan en slechts dan als er een toekenning van waarden bestaat aan de Boole'se variabelen U, zodanig dat aan elke claus uit C tegelijkertijd is voldaan. Aan bijvoorbeeld $C = \{\{u_1, u_2, u_3\}, \{\bar{u}_2, \bar{u}_3\}, \{u_2, \bar{u}_3\}\}$ kan worden voldaan omdat aan iedere claus in C wordt voldaan door ieder van de volgende waardetoekenningen.

u_1	T	T	F
u_2	T	F	T
u_3	F	F	F

Aan de volgende verzameling clausen kan niet worden voldaan $\{\{u_1, u_2\}, \{u_1, \bar{u}_2\}, \{\bar{u}_1\}\}$.

Het beslissingsprobleem kan zo worden geformuleerd

Vervulbaarheidsprobleem (SAT)

Gegeven: Een eindige verzameling U van Boole'se variabelen en een eindige verzameling C van clausen over U.

Gevraagd: Kan aan C door een waardetoekenning worden vervuld?

Cook (1971) formuleerde de volgende stelling die aanleiding heeft gegeven tot een groot aantal belangrijke resultaten.

Stelling 6.6 (Stelling van Cook)

SAT is NP-compleet.

Bewijs. Om te bewijzen dat SAT $\in NPC$ moeten we aantonen dat SAT $\in NP$ en dat voor alle $\Pi' \in NP$ geldt dat $\Pi' \propto$ SAT.

Dat SAT $\in NP$ is vrij duidelijk. Als een instantie I van SAT n Boole'se variabelen bevat dan raadt het nondeterministische programma

dat I oplost eenvoudig naar een oplossing door waarden aan de variabelen toe te kennen en het controleert vervolgens of deze waardetoekenning maakt dat alle clausen zijn vervuld. Als niet alle clausen kunnen worden vervuld dan zal via raden geen oplossing worden gevonden.

Het lijkt op het eerste gezicht vrij moeilijk om te bewijzen dat voor alle $\Pi' \in NP$ geldt dat $\Pi' \propto \text{SAT}$. We weten per slot van rekening niet eens welke problemen er in NP zitten! Het enige dat we weten is dat er bij ieder probleem Π' een NDTM, M' moet bestaan die het probleem in polynomiale tijd oplost. Van dit gegeven moeten we dan ook uitgaan. Zij $M' = (Q,\Sigma,T,q_0,F)$ een NDTM die $L' = L_{\Pi',e} \subseteq T^*$ accepteert bij een geschikte codering e'. We mogen veronderstellen dat de tijdcomplexiteitsfunctie voor deze NDTM $0(n^k)$ is voor een positief geheel getal k. Verder kunnen we op grond van stelling 6.2 veronderstellen dat M' graad 2 heeft.

Omdat $\text{SAT} \in NP$ weten we dat er een NDTM van polynomiale tijd is, M_{SAT} die $L = L_{\text{SAT},e}$ accepteert bij een geschikte codering e. Als we uitgaan van het taalmodel dan moeten we aantonen dat $L' \propto L$ en dat dientengevolge op probleemniveau geldt dat $\Pi' \propto \text{SAT}$. We zullen voor elke L' laten zien hoe we een functie $f_{L'}$ kunnen construeren die strings in T^* afbeeldt naar instanties van SAT. Deze instanties zouden dan kunnen worden gecodeerd met behulp van e om zo strings in L te verkrijgen. Onze functie $f_{L'}$ zal voldoen aan $x \in L'$ desd als $f_{L'}(x)$ een waardetoekenning heeft die voldoet (en als zijn codering zich dus in L bevindt). Zodra we dus $f_{L'}$ hebben bepaald en hebben aangetoond dat deze functie in polynomiale tijd berekenbaar is, hebben we het bestaan van de gewenste polynomiale transformatie bewezen. We laten nu zien hoe $f_{L'}(x)$ wordt gedefinieerd in termen van de NDTM, M'.

Als $x \in T^*$ door M' wordt geaccepteerd dan moet deze acceptatie plaatsvinden in een tijd $\leqslant m = \max(c_1, c_2|x^k|)$ voor bepaalde constanten c_1 en c_2, omdat de tijdcomplexiteitsfunctie immers $0(n^k)$ is. Dit wil zeggen dat van de door de machine te gebruiken tape alleen het stuk tussen positie $-m+1$ en $m+1$ wordt gebruikt om te testen of een gegeven input $x \in T^*$ wordt geaccepteerd. De configuratie van M' in ieder stadium van de berekening kan volledig worden gespecificeerd met behulp van de inhouden van deze $2m+1$ locaties in combinatie met de huidige toestand en de positie van de lees/schrijfkop. Op

grond van stelling 6.1 mogen we veronderstellen dat M' maar een acceptatietoestand heeft en de toestandverzameling Q bevat dus q_0 (de begintoestand), q_1 (de acceptatietoestand) en $q_2,q_3,\ldots,q_a$ voor een $a \geqslant 1$, De verzameling symbolen die op de tape kunnen staan is Σ en bestaat uit $s_0 = \wedge, s_1, s_2, \ldots, s_b$ voor een $b \geqslant |T|+1$. We definiëren nu de volgende Boole'se variabelen.

Toestand-variabelen: voor alle $0 \leqslant i \leqslant m$, $0 \leqslant k \leqslant a$ een variabele $Q[i,k]$ die de waarde 'waar' krijgt desd als M' zich in toestand q_k bevindt op tijdstip i.

Kop-variabelen: voor alle $0 \leqslant i \leqslant m$, $-m+1 \leqslant j \leqslant m+1$ een variabele $H[i,j]$ die de waarde 'waar' krijgt desd als de tapekop op tijdstip i de locatie j leest.

Symbool-variabelen: voor alle $0 \leqslant i \leqslant m$, $-m+1 \leqslant j \leqslant m+1$, $0 \leqslant k \leqslant b$ een variabele $S[i,j,k]$ die de waarde 'waar' krijgt desd als het symbool in locatie j op tijdstip i gelijk aan s_k is.

Keuze-variabelen: voor alle $0 \leqslant k \leqslant a$, $0 \leqslant l \leqslant b$ zodanig dat $\#P(q_k, s_1) = 2$ en voor alle $0 \leqslant i \leqslant m$ en $-m+1 \leqslant j \leqslant m+1$ twee Boole'se variabelen $P_1[i,j,k,l]$ en $P_2[i,j,k,l]$ die worden gebruikt om aan te geven welke van de twee mogelijke stappen wordt gekozen.

Al deze variabelen tezamen vormen de verzameling U voor het SAT-probleem dat we uit M' en een gegeven $x \in T^*$ construeren. Nu moeten we de bijbehorende clausen nog construeren. Deze construeren we uit de string x op een zodanige wijze dat aan alle clausen voldaan kan worden dan en slechts dan als er een berekening door M' is die x accepteert. Aan ons SAT-probleem kan dus voldaan worden desd als $x \in L'$.

De clausen worden in verschillende groepen geconstrueerd met behulp van de volgende regels.

Unieke toestand

Op ieder tijdstip $0 \leqslant i \leqslant m$ moet de NDTM, M' zich in een unieke (dat wil zeggen in een en slechts een) toestand bevinden. De clausen

$$\{Q[i,0], Q[i,1], \ldots, Q[i,a]\} \quad \text{voor alle} \quad 0 \leqslant i \leqslant m$$

zorgen ervoor dat de NDTM zich op elk moment in minstens één toestand bevindt. De clausen

$\{\overline{Q[i,j]},\overline{Q[i,j']}\}$ voor alle $0 \leqslant i \leqslant m$ en $0 \leqslant j \leqslant j' \leqslant a$

verzekeren ons ervan dat die toestand uniek is.

Unieke koppositie

Om ervoor te zorgen dat de kop op elk tijdstip een unieke locatie leest gebruiken we de clausen

$\{H[i,-m+1],H[i,-m+2],\ldots,H[i,m+1]\}$ voor alle $0 \leqslant i \leqslant m$

en

$\{\overline{H[i,j]},\overline{H[i,j']}\}$ voor alle $0 \leqslant i \leqslant m$, $-m+1 \leqslant j < j' \leqslant m+1$

Uniek symbool

De volgende clausen gebruiken we om ervoor te zorgen dat in elke locatie op elk tijdstip een uniek symbool staat

$\{S[i,j,0],S[i,j,1],\ldots,S[i,j,b]\}$ voor alle $0 \leqslant i \leqslant m$
en $-m+1 \leqslant j \leqslant m+1$

en

$\{\overline{S[i,j,k]},\overline{S[i,j,k']}\}$ voor alle $0 \leqslant i \leqslant m$, $-m+1 \leqslant j \leqslant m+1$
en $0 \leqslant k < k' \leqslant b$

Initiële configuratie

Op tijdstip 0 wordt $x \in T^*$ geschreven in de locaties $1,2,\ldots,|x|$. Alle andere locaties bevatten het lege symbool; de tapekop leest locatie 1 en de huidige toestand is q_0. Vertaald in clausen krijgen we

$\{Q[0,0]\}$
$\{H[0,1]\}$
$\{S[0,1,i_1]\},\{S[0,2,i_2]\},\ldots,\{S[0,|x|,i_{|x|}]\}$ met $x = s_{i_1}s_{i_2}\ldots s_{i_{|x|}}$
en $\{S[0,j,0]\}$ voor $-m+1 \leqslant j \leqslant 0$ en $|x| < j \leqslant m+1$

Eind-acceptatieconfiguratie

Als geldt dat $x \in L'$ dan moet de NDTM op een tijdstip $\leqslant m$ de string x hebben geaccepteerd en de machine zal zich dan dus in toestand q_1 bevinden. Hier kunnen we voor zorgen door te eisen dat als $Q[i,1]$ waar is dat dan ook $Q[i+1,1]$ waar moet zijn. Verder moet ook $Q[m,1]$ waar zijn en dus construeren we de volgende clausen

$\{\overline{Q[i,1]}, Q[i+1,1]\}$ voor alle $0 \leqslant i \leqslant m$

en

$\{Q[m,1]\}$

Configuratie-overgangen

We voeren tenslotte een groep clausen in die ervoor zorgen dat de configuratie-overgangen volgens de regels van de machine M' verlopen. Stel dat de tapekop op tijdstip i locatie j leest, zich in toestand q_k bevindt en dat het symbool in locatie j gelijk is aan s_j. Dit betekent dat $H[i,j], Q[i,k]$ en $S[i,j,l]$ alle waar moeten zijn.

Als $P(q_k, s_l)$ is gedefinieerd dan kunnen we twee gevallen onderscheiden.

Geval 1. $\#P(q_k,s_l) = 1$ dus $P(q_k,s_l) = \{(q_{k'}, s_{l'}, X)\}$ voor een $q_{k'} \in Q$, $s_{l'} \in \Sigma$ en $X \in \{L,0,R\}$. Als $X = L$ dan willen we dat $H[i+1,j-1]$, $Q[i+1,k']$ en $S[i+1,j,l']$ waar zijn en we voeren daarom de volgende clausen in

$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, H[i+1,j-1]\}$$

$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, Q[i+1,k']\}$$

en $\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, S[i+1,l']\}$

Als echter $X = 0$ dan vervangen we de eerste claus door

$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, H[i+1,j]\}$$

en als $X = R$ door

$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, H[i+1,j+1]\}$$

De andere twee clausen blijven onveranderd.

Geval 2. $\#P(q_k,s_l) = 2$ dus $P(q_k,s_l) = (q_{k'}, s_{l'}, X), (q_{k''}, s_{l''}, Y)$ voor $q_{k'}, q_{k''} \in Q$, $s_{l'}, s_{l''} \in \Sigma$ en $X, Y \in \{L,0,R\}$. We hebben voor deze gevallen de twee volgende Boole'se variabelen nodig: $P_1[i,j,k,l]$ en $P_2[i,j,k,l]$. Vervolgens introduceren we de clausen

$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, P_1[i,j,k,l], P_2[i,j,k,l]\}$$

en

$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, \overline{P_1[i,j,k,l]}, \overline{P_2[i,j,k,l]}\}$$

Op deze wijze zijn we er zeker van dat als $H[i,j]$, $Q[i,k]$ en $S[i,j,l]$ waar zijn dat dan precies een van de literalen $P_1[i,j,k,l]$, $P_2[i,j,k,l]$ waar is. Als $P_1[i,j,k,l]$ waar is dan selecteren we $(q_{k'}, s_{l'}, X)$ als volgende stap en als $P_2[i,j,k,l]$ waar is dan kiezen we $(q_{k''}, s_{l''}, Y)$. De toestandsverondering representeren we door

$$\{\overline{P_1[i,j,k,l]}, Q[i+1,k']\}$$

en

$$\{\overline{P_2[i,j,k,l]}, Q[i+1,k'']\}$$

Een verandering van symbool wordt gedekt door de clausen

$$\{\overline{P_1[i,j,k,l]}, S[i+1,j,l']\}$$

en

$$\{\overline{P_2[i,j,k,l]}, S[i+1,j,l'']\}$$

Tenslotte volgt nog de richting van de stap. Hiervoor gebruiken we

	$\{\overline{P_1[i,j,k,l]}, H[i+1,j-1]\}$	als	$X = L$
of	$\{\overline{P_1[i,j,k,l]}, H[i+1,j]\}$	als	$X = 0$
of	$\{\overline{P_1[i,j,k,l]}, H[i+1,j+1]\}$	als	$X = R$

samen met een analoge claus

	$\{\overline{P_2[i,j,k,l]}, H[i+1,j-1]\}$	als	$Y = L$
of	$\{\overline{P_2[i,j,k,l]}, H[i+1,j]\}$	als	$Y = 0$
of	$\{\overline{P_2[i,j,k,l]}, H[i+1,j+1]\}$	als	$Y = R$

Hiermee zijn we klaar met geval 2.

De voor de bovengenoemde twee gevallen geconstrueerde clausen verzekeren ons ervan dat er clausen bestaan voor de volgende stap. We moeten er echter ook voor zorgen dat het symbool in locatie j niet verandert tussen tijdstip i en $i+1$ als de lees/schrijfkop op tijdstip i *niet* boven locatie j staat. Hiervoor voeren we de volgende claus in

$$\{\overline{S[i,j,l]}, H[i,j], S[i+1,j,l]\}$$

voor alle $0 \leqslant i \leqslant m$, $-m+1 \leqslant j \leqslant m+1$ en $0 \leqslant l \leqslant b$.

De constructie van de benodigde clausen is nu gereed. Als $x \in L'$ dan bestaat er een berekening door machine M' voor de acceptatie van x die tijd $\leqslant m$ in beslag neemt. Gegeven de interpretatie van de in het voorgaande gedefinieerde Boole'se variabelen kent deze berekening waarden toe zodanig dat aan alle clausen die we construeerden is voldaan. Het is zelfs zo dat de constructie van de clausen ons ervan verzekert dat iedere voldoende waardetoekenning overeenkomt met een berekening door M' die x accepteert. Hieruit volgt dat aan $f_{L'}(x)$ (de verzameling clausen die we uit de string x construeerden) kan worden voldaan desd $x \in L'$. Als u verder het aantal gebruikte Boole'se variabelen en het aantal clausen telt (zie oefening 6.6) dan zult u zien dat beide aantallen worden begrensd door polynomen in $|x|$. (Bedenk hierbij dat a en b constanten zijn.) Uit dit alles mogen we de gevolgtrekking maken dat $f_{L'}(x)$ kan worden berekend door een algoritme van polynomiale tijd. We hebben nu dus aangetoond dat er voor elke $L' \in NP$ een polynomiale transformatie bestaat van L' naar SAT,*e*. SAT is dus NP-compleet. Merk nog op dat we voor het leveren van dit bewijs niet precies hoefden te weten hoe de DNTM die L' accepteert werkt; we behoefden alleen maar te weten dat zo'n machine bestaat.

Nu we het bestaan van een NP-compleet probleem hebben aangetoond, kunnen we de volgende stelling gebruiken om er veel meer te vinden.

Stelling 6.7

Als L NP-compleet is en $L_1 \in NP$ dan volgt uit $L \propto L_1$ dat ook L_1 NP-compleet is.

Bewijs. Voor iedere $L' \in NP$ geldt dat $L' \propto L$ op grond van de definitie van NP-compleetheid. Ook geldt $L \propto L_1$ en op grond van stelling 6.3 is $\propto$ transitief. Dus geldt voor elke $L' \in NP$ dat $L' \propto L_1$ en omdat $L_1 \in NP$ is hiermee het bewijs geleverd.

We hebben nu een strategie voor het formuleren van een hele rij NP-complete problemen. Allereerst weten we dat SAT $\in NPC$. Verder

weten we dat als $\Pi \in NPC$ en $\Pi_1 \in NP$ en we kunnen aantonen dat $\Pi \propto \Pi_1$ dat hieruit volgt dat $\Pi_1 \in NPC$. Dus om precies te zijn: als $\Pi_1 \in NP$ en SAT $\propto \Pi_1$, dan moet ook Π_1 *NP*-compleet zijn. SAT is het zaadje waaruit alle nu bekende *NP*-complete problemen zijn voortgekomen. Op dit moment zijn er een paar honderd van dergelijke problemen bekend. Een overzicht van de resultaten bekend in 1978 staat in *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Garey & Johnson, 1978). Sindsdien wordt de lijst regelmatig aangevuld via een artikelenreeks in het *Journal of Algorithms*. In de volgende paragraaf geven we enkele voorbeelden van *NP*-complete problemen en de bijbehorende bewijzen. Een en ander ter illustratie van het soort bewijzen dat wordt gebruikt en van de uiteenlopende aard van de problemen in de klasse *NPC*. Zelfs als we aannemen dat $P \neq NP$ dan moet men nog niet veronderstellen dat elk probleem in *NP* ofwel in *P* zit ofwel *NP*-compleet is. Het is zelfs bewezen dat als $P \neq NP$, dat dan $NPI = NP \backslash (P \cup NPC)$ een oneindige collectie van verschillende equivalentieklassen bevat onder polynomiale equivalentie (zie Ladner, 1975). Toegegeven moet echter worden dat de grote meerderheid van in de praktijk voorkomende problemen inderdaad in *P* of in *NPC* liggen. De volgende twee problemen liggen misschien in *NPI*, maar dit is slechts een veronderstelling.

Graaf-isomorfisme

Gegeven: Twee grafen G_1 en G_2.
Gevraagd: Is G_1 isomorf met G_2?

Samengestelde getallen

Gegeven: Een positief geheel getal k.
Gevraagd: Bestaan er gehele getallen m en $n > 1$ zodat $k = mn$?

Onder bepaalde voorwaarden (bijvoorbeeld bij twee planaire grafen) is graaf-isomorfisme in polynomiale tijd oplosbaar. Voor het algemene geval is echter geen polynomiaal algoritme bekend. Ook heeft niemand nog kunnen aantonen dat het hier om een *NP*-compleet probleem gaat, hoewel van *subgraaf-isomorfisme* (Is G_1 isomorf met een subgraaf van G_2?) wel bekend is dat het een *NP*-compleet probleem is (zie oefening 6.11).

Als we een deterministisch algoritme zouden kunnen ontwerpen dat in polynomiale tijd test of een bepaald getal priem is dan zou het probleem *samengesteld getal* in *P* liggen. Een dergelijk algoritme voor het testen op primaliteit bestaat nog niet, maar er zijn wel verbazend goede resultaten bereikt met behulp van een waarschijnlijkheidstheoretische aanpak. Het artikel van M. Rabin 'Probabilistic algorithms' (Traub, 1976), beschrijft een stochastisch algoritme voor het vaststellen van primaliteit dat werkt in polynomiale tijd en dat slechts een kleine kans heeft op een onjuiste beslissing gegeven een willekeurige input. Stochastische algoritmen loten in feite welke optie uit een aantal mogelijke opties gekozen wordt. Het moet dan wel zo zijn dat de meerderheid van de mogelijke te volgen paden tot een juist antwoord leidt. *Monte-Carlo algoritmen* stoppen altijd maar hebben een kleine kans op een onjuist antwoord. *Las Vegas 'algoritmen'* geven altijd het juiste antwoord maar lopen een kleine kans niet te stoppen. *Atlantic City 'algoritmen'* stoppen soms niet en als zij stoppen is er nog een kans dat het antwoord niet correct is. De kracht van stochastische algoritmen ligt in het feit dat zij kunnen worden herhaald zodat de kans op een fout willekeurig klein kan worden gemaakt. Een theoretische behandeling van stochastische algoritmen wordt gegeven in Gill (1977).

EEN BLOEMLEZING VAN NP-COMPLETE PROBLEMEN

Allereerst behandelen we een restrictie van SAT, waarbij alle clausen precies drie literalen bevatten.

3-Vervulbaarheidsprobleem (3SAT)
Gegeven: Een verzameling U van Boole'se variabelen en een verzameling C van clausen over U, waarbij elke $c \in C$ zodanig is dat $\#(c) = 3$ en dus drie literalen bevat.
Gevraagd: Bestaat er een waardetoekenning die C vervult?

Stelling 6.8

3SAT is *NP*-compleet.

Bewijs. Er geldt 3SAT $\in NP$ omdat we gemakkelijk een willekeurig geraden waardetoekenning in polynomiale tijd kunnen controleren. We tonen aan dat 3SAT $\in NPC$ door een polynomiale transformatie SAT $\propto$ 3SAT aan te geven.

Zij $U = \{u_1, u_2, \ldots, u_n\}$ en $C = \{c_1, c_2, \ldots, c_m\}$ een instantie I van SAT. We construeren nu een instantie $f(I)$ van 3SAT zodanig dat
(i) $I \in Y_{\text{SAT}}$ desd als $f(I) \in Y_{\text{3SAT}}$ en
(ii) f is berekenbaar in polynomiale tijd.
Om $f(I)$ te construeren nemen we eenvoudig elke claus $c_i \in C$ en vervangen die door een gelijkwaardige verzameling van 3-literaal clausen. Hiervoor hebben we enkele extra Boole'se variabelen nodig; de Boole'se variabelen die worden gebruikt bij de constructie van de clausen die overeenkomen met c_i liggen in de verzameling U_i'. De verzameling U' omvat dan alle extra Boole'se variabelen die we op deze wijze toevoegen.

Stel nu dat $c_i \in C$. We moeten dan vier gevallen onderscheiden.

Geval 1. $\#(c_i) = 1$. Dus bijvoorbeeld $c_i = x_1$.
Dan is $U_i' = \{u_i', u_i''\}$, waarbij u_i' en u_i'' nieuwe Boole'se variabelen zijn. We vervangen nu c_i door de volgende vier clausen.

$$\{x_1, u_i', u_i''\}, \{x_1, u_i', \bar{u}_i''\}, \{x_1, \bar{u}_i', u_i''\}, \{x_1, \bar{u}_i', \bar{u}_i''\}.$$

Geval 2. $\#(c_i) = 2$, dus bijvoorbeeld $c_i = \{x_1, x_2\}$.
Dan is $U_i' = \{u_i'\}$ waarbij u_i' een nieuwe Boole'se variabele is.
We vervangen c_i door de volgende twee clausen

$$\{x_1, x_2, u_i'\}, \{x_1, x_2, \bar{u}_i'\}.$$

Geval 3. $\#(c_i) = 3$ dus we hoeven niets te doen en $U_i' = \phi$.

Geval 4. $\#(c_i) = k > 3$, dus $c_i = \{x_1, x_2, \ldots, x_k\}$.
Nu zijn $U_i = u_i^j \, |1 \leqslant j \leqslant k-3\}$ alle nieuwe variabelen.
We vervangen c_i door de clausen

$$\{x_1, x_2, u_i^1\}$$
$$\{\bar{u}_i^1, x_3, u_i^2\}$$
$$\{\bar{u}_i^2, x_4, u_i^2\}$$
$$\ldots$$
$$\{\bar{u}_i^{k-3}, x_{k-1}, x_k\}.$$

Merk op dat een claus in C met k literalen zowel u als $\bar{u}$ moet bevatten voor een of andere $u \in U$ als $k > n$. De claus is dan dus triviaal vervuld door iedere waardetoekenning en deze kan verder worden gegenereerd. We kunnen daarom dus aannemen dat $\#(c_i) \leqslant n$ voor $1 \leqslant i \leqslant m$ en zo zijn dus zowel het aantal nieuwe variabelen als het aantal clausen dat we hierboven construeerden begrensd door polynomen in mn. Dus is f een polynomiale transformatie.

Stel nu dat $t: U \rightarrow \{T,F\}$ een toekenning is van logische waarden aan de variabelen van U zodanig dat iedere claus in C vervuld is. We kunnen t dan uitbreiden tot een waardetoekenning $t: U \cup U' \rightarrow \{T,F\}$ die voldoet aan alle geconstrueerde clausen. Eerst laten we zien hoe we t kunnen laten werken op een willekeurige verzameling in U_i'. Als U_i' werd geconstrueerd zoals in de bovenvermelde gevallen 1, 2 of 3 dan is elke uitbreiding van t geschikt omdat t zelf al voldoende is. Werd U_i' geconstrueerd zoals in geval 4 dan weten we dat er enige x_l, $1 \leqslant l \leqslant k$ moeten zijn geweest met $t(x_l) = T$. Als $l = 1$ of $l = 2$ dan nemen we $t(u_i^j) = F$ voor $1 \leqslant j \leqslant k-3$. Is $l = k-1$ of $l = k$ dan nemen we $t(u_i^j) = T$ voor $1 \leqslant j \leqslant k-3$. In alle andere gevallen nemen we $t(u_i^j) = T$ voor $1 \leqslant j \leqslant l-2$ en $t(u_i^j) = F$ voor $l-1 \leqslant j \leqslant k-3$. We breiden de werking van t dus uit op elk der verzamelingen U_i', $i = 1,2,\ldots,m$ en vinden zo dus een toekenning van logische waarden $t: U \cup U' \rightarrow \{T,F\}$ die alle geconstrueerde clausen vervult. Dus $I \in Y_{\text{SAT}}$ impliceert dat $f(I) \in Y_{\text{SAT}}$. Het omgekeerde is veel gemakkelijker te bewijzen. Bij elke waardetoekenning die alle geconstrueerde clausen vervult kan gemakkelijk worden aangetoond dat de restrictie tot U de oorspronkelijke clausen vervult. Dus $I \in Y_{\text{SAT}}$ desd als $f(I) \in Y_{\text{SAT}}$. Hiermee is het bewijs geleverd.

Ofschoon 3SAT dus NP-compleet is geldt dit niet voor 2SAT (elke claus heeft precies twee literalen) want dit laatste probleem is in polynomiale tijd oplosbaar (oefening 6.7). Dit is een vrij algemeen verschijnsel: vaak is een probleem ten aanzien van tripels NP-compleet, terwijl hetzelfde probleem ten aanzien van paren in P ligt. (Misschien zijn er daarom maar twee geslachten!) Een illustratie van dit feit vormt ook het volgende probleem, waarvan bekend is dat het NP-compleet is. (Het bewijs hiervan tezamen met vele andere bewijzen wordt gegeven in het

baanbrekende artikel door R.M. Karp, 'Reducibility among combinatorial problems' [Miller & Thatcher, 1972].)

3-dimensionaal matching (3DM)

Gegeven: Een verzameling $M \subseteq X \times Y \times Z$ waarbij X, Y en Z disjuncte verzamelingen zijn met elk n elementen.

Gevraagd: Bevat M een matching, dat wil zeggen een deelverzameling $M' \subseteq M$ zodanig dat $|M'| = n$ en zo dat de coördinaten van de elementen van M' stuk voor stuk onderling verschillend zijn?

Bijvoorbeeld als $X = \{x_1,x_2\}$, $Y = \{y_1,y_2\}$ en $Z = \{z_1,z_2\}$ dan bevat

$$\{(x_1,y_2,z_1),\\ (x_1,y_1,z_1),\\ (x_1,y_2,z_1),\\ (x_2,y_1,z_2)\}$$

een matching, namelijk (x_1,y_2,z_1) en (x_2,y_1,z_2), maar bijvoorbeeld $\{(x_1,y_1,z_1),(x_1,y_2,z_1),(x_2,y_2,z_1)\}$ bevat er geen.

Het overeenkomstige *2-dimensionale matching probleem* ligt in P (zie Hopcroft & Karp, 1973).

De rest van deze paragraaf zullen we besteden aan het aantonen van de NP-compleetheid van het *Hamilton circuit probleem*, (HC). We zagen al dat HC $\propto$ TSP en dat TSP $\in NP$. Een gevolg van ons bewijs zal dus zijn dat ook TSP NP-compleet is. Om aan te tonen dat HC $\in NPC$ is behoeven we alleen maar te laten zien dat HC $\in NP$ en vervolgens moeten we een NP-compleet probleem Π vinden zodanig dat $\Pi \propto$ HC. Hiervoor kunnen we elke $\Pi \in NPC$ gebruiken; op grond van de theorie weten we dat als HC $\in NPC$ er altijd een transformatie bestaat. De grote kunst bij dit soort bewijzen is een bekend NP-compleet probleem te kiezen dat het te leveren bewijs zo eenvoudig mogelijk maakt. We zullen een indirect bewijs geven via een probleem dat bekend staat als het *knopenoverdekkingsprobleem* (Engels: Vertex cover).

Knopenoverdekking (VC)

Gegeven: Een graaf met knopenverzameling V en kantenverzameling E en een positief geheel getal $b \leqslant \#(V)$.

Gevraagd: Bestaat er een knopenoverdekking met omvang kleiner of gelijk b voor G? Dat wil zeggen bestaat er een deelverzameling $V' \subseteq V$ zodanig dat $\#(V') \leqslant b$ en zo dat voor elke kant $u \overset{e}{\text{———}} v$ in V geldt dat minstens een der knopen u en v tot V' behoort?

We zullen laten zien dat 3SAT $\propto$ VC en dat VC $\propto$ HC. Naarmate het bestand van bekende NP-complete problemen groeit, worden bewijzen gemakkelijker omdat de kans toeneemt dat men een bekend probleem $\Pi \in NPC$ kan vinden dat gemakkelijk door middel van een polynomiale transformatie naar het beschouwde probleem kan worden omgezet. Voor onderzoekers op dit gebied is het dus van belang op de hoogte te blijven van de nieuwste ontwikkelingen. In dit boek geven we slechts een eerste idee van het soort bewijzen dat wordt gebruikt en van de resultaten die tot nu toe zijn bereikt.

Stelling 6.9

VC is NP-compleet.

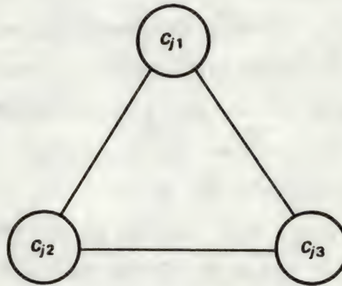
Bewijs. Omdat we alleen maar naar een deelverzameling van V met omvang b hoeven te raden en vervolgens kunnen controleren of van elke kant in E minstens een der knopen in deze deelverzameling ligt, geldt VC $\in NP$. Het genereren van de deelverzameling en de controle kunnen gemakkelijk in polynomiale tijd worden uitgevoerd.

We geven nu een polynomiale transformatie om te bewijzen dat VC $\in NPC$. Stel dat I een instantie van 3SAT is die bestaat uit de verzameling Boole'se variabelen $U = \{u_1, u_2, \ldots, u_n\}$ en uit de verzameling $C = \{c_1, c_2, \ldots, c_m\}$ van clausen van drie literalen. Uit I moeten we nu een functie $f(I)$ construeren die leidt tot een graaf G en een grens b, zodanig dat $f(I) \in Y_{\mathrm{VC}}$, dus zo dat G een knopenoverdekking $\leqslant b$ bezit desd als $I \in Y_{\mathrm{SAT}}$. Daarbij moeten we dan ook nog aantonen dat deze constructie in polynomiale tijd kan worden uitgevoerd.

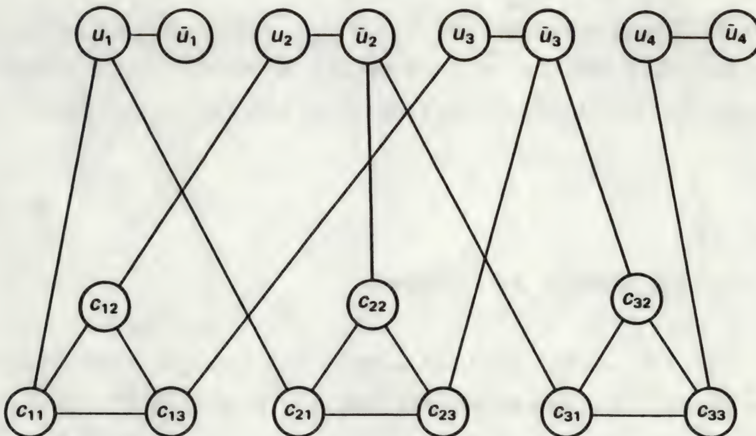
Voor elke $u_i \in U$ construeren we twee knopen u_i en u_j, onderling verbonden door een enkele kant, u_i —— u_j.

Elke claus $c_j \in C$ gaat over in drie knooppunten c_{j1}, c_{j2}, c_{j3}, onderling verbonden zoals in figuur 6.4 is weergegeven. Als nu bijvoorbeeld $c_j = \{x, y, z\}$, waarbij x, y en z literalen voorstellen, dan verbinden we knoop c_{j1} met x, c_{j2} met y en c_{j3} met z.

Op deze manier construeren we een graaf die overeenkomt met I. In figuur 6.5 hebben we bijvoorbeeld de graaf weergegeven die overeenkomt met een 3SAT probleem dat bestaat uit $U = \{u_1, u_2, u_3, u_4\}$ en $C = \{u_1, u_2, u_3\}, \{u_1, \bar{u}_2, \bar{u}_3\}, \{\bar{u}_2, \bar{u}_3, u_4\}\}$. De grens b is dan gelijk aan $n+2m$.



Figuur 6.4



Figuur 6.5

Stel nu dat er een voldoende waardetoekenning $t: U \to \{T, F\}$ voor I bestaat. We kunnen dan als volgt een knopenoverdekking met omvang $n+2m$ voor G selecteren. Kies voor elke $u_i \in U$, u_i als $t(u_i) = T$ en kies $\bar{u}_i$ als $t(u_i) = F$, (zo krijgen we n knopen). Omdat nu C vervuld wordt door de waarden die t aan de logische variabelen toekent, moet voor elke $c_i \in C$ een der knopen c_{j1}, c_{j2}, c_{j3} verbonden zijn met een geselecteerde knoop. We nemen daarom de andere twee knopen ook in de overdekking op en krijgen zo een overdekking met omvang $n+2m$. Hiermee hebben we aangetoond dat $I \in Y_{\mathrm{SAT}} \Rightarrow f(I) \in Y_{\mathrm{VC}}$.

Nu het bewijs in omgekeerde richting. Neem hiervoor een willekeurige knopenoverdekking met omvang $b \leqslant n+2m$. Omdat er steeds een directe verbinding bestaat tussen u_i en $\bar{u}_i$ voor elke $u_i \in U$, moet deze knopenoverdekking minstens één van deze twee knopen bevatten. Net zo geldt omdat c_{j1}, c_{j2}, c_{j3} in een driehoek liggen dat minstens twee van deze knopen in de overdekking moeten zitten. Dit betekent dat elke overdekking minstens uit $n+2m$ knopen bestaat en een overdekking met hoogstens $n+2m$ knopen bevat er dus precies $n+2m$ en bevat dan precies één knoop uit elk paar $u_i, \bar{u}_i$. We definiëren nu een waardetoekenning door $t(u_i) = T$ als u_i in de overdekking zit en door $t(u_i) = F$ als dat niet het geval is. Deze toekenning is voldoende voor elke claus $c_j \in C$. Immers slechts twee van de knopen c_{j1}, c_{j2}, c_{j3} zitten in de overdekking en een daarvan moet dus een kant hebben naar een geselecteerde u_i of $\bar{u}_i$. Onder t is dus iedere c_j vervuld en dus geldt $f(I) \in Y_{\mathrm{VC}} \Rightarrow I \in Y_{\mathrm{SAT}}$.

Nu we hebben aangetoond dat $I \in Y_{\mathrm{SAT}}$ desd als $f(I) \in Y_{\mathrm{VC}}$ hoeven we alleen nog maar op te merken dat de constructie duidelijk in polynomiale tijd kan worden uitgevoerd en hiermee is het bewijs gereed.

Stelling 6.10

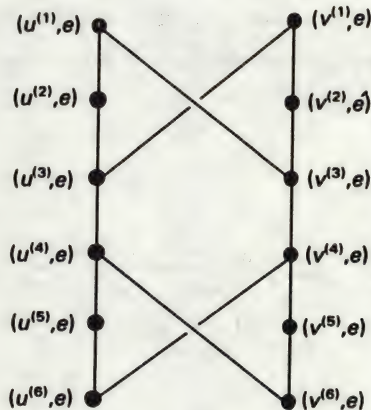
HC (en dus ook TSP) is NP-compleet.

Bewijs. HC $\in NP$ omdat men met behulp van een nondeterministisch algoritme alleen maar een of andere permutatie van de n knopen $v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n)}$ van de gegeven graaf behoeft te raden en vervolgens in polynomiale tijd kan checken of er een in zichzelf terug-

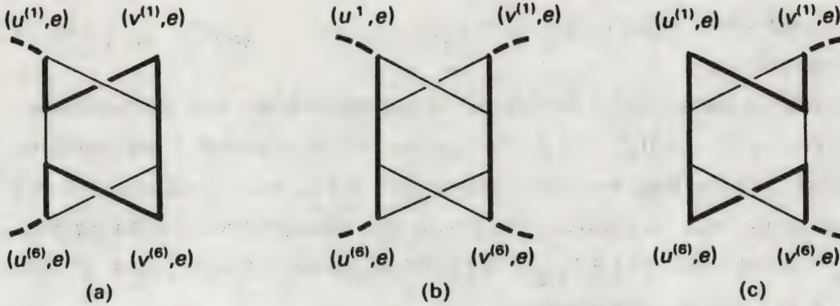
kerend pad (een cykel) $v_{\sigma(1)} - v_{\sigma(2)} - \ldots - v_{\sigma(n)} - v_{\sigma(1)}$ in de graaf voorkomt.

Om aan te tonen dat $HC \in NPC$ construeren we een polynomiale transformatie $VC \propto HC$. Zij I, bestaande uit een graaf G met knopen V en kanten E te zamen met een grens b $\#(V)$, een instantie van VC. We moeten nu een instantie $f(I)$ van HC construeren, zodanig dat $I \in Y_{VC}$ desd als $f(I) \in Y_{HC}$. $f(I)$ zal bestaan uit een graaf G' die we als volgt uit G construeren.

Om te beginnen voorzien we de graaf G' van b knopen $w_1, w_2, \ldots, w_b$ die we zullen gebruiken om b knopen uit de knopenverzameling V te kiezen. Voor elke kant $e \in E$ bestaat er nu een subgraaf G'_e van G' zodanig dat we er zeker van zijn dat minstens één van de twee eindpunten van de kant e voorkomt onder de b geselecteerde knopen. Veronderstel dat e samenvalt met de kant tussen de knopen u en $v \in V$ dan construeren we G'_e zo dat in deze graaf 12 knopen $(u^{(i)},e),(v^{(i)},e)$, $i = 1,2,\ldots,6$ voorkomen die onderling door 14 kanten zijn verbonden zoals in figuur 6.6 is aangegeven. De knopen $(u^{(1)},e),(u^{(6)},e)$, $(v^{(1)},e)(v^{(6)},e)$ heten de *hoekknopen* van de subgraaf en alleen deze knopen zullen we verbinden met andere stukken van de graaf G'. Ieder Hamiltoncircuit binnen G' dient dan een van de configuraties uit figuur 6.7 te bevatten.



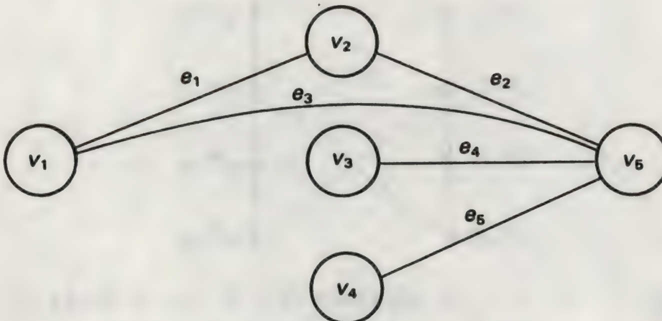
Figuur 6.6 G', de subgraaf van G' die behoort bij de kant e tussen u en v



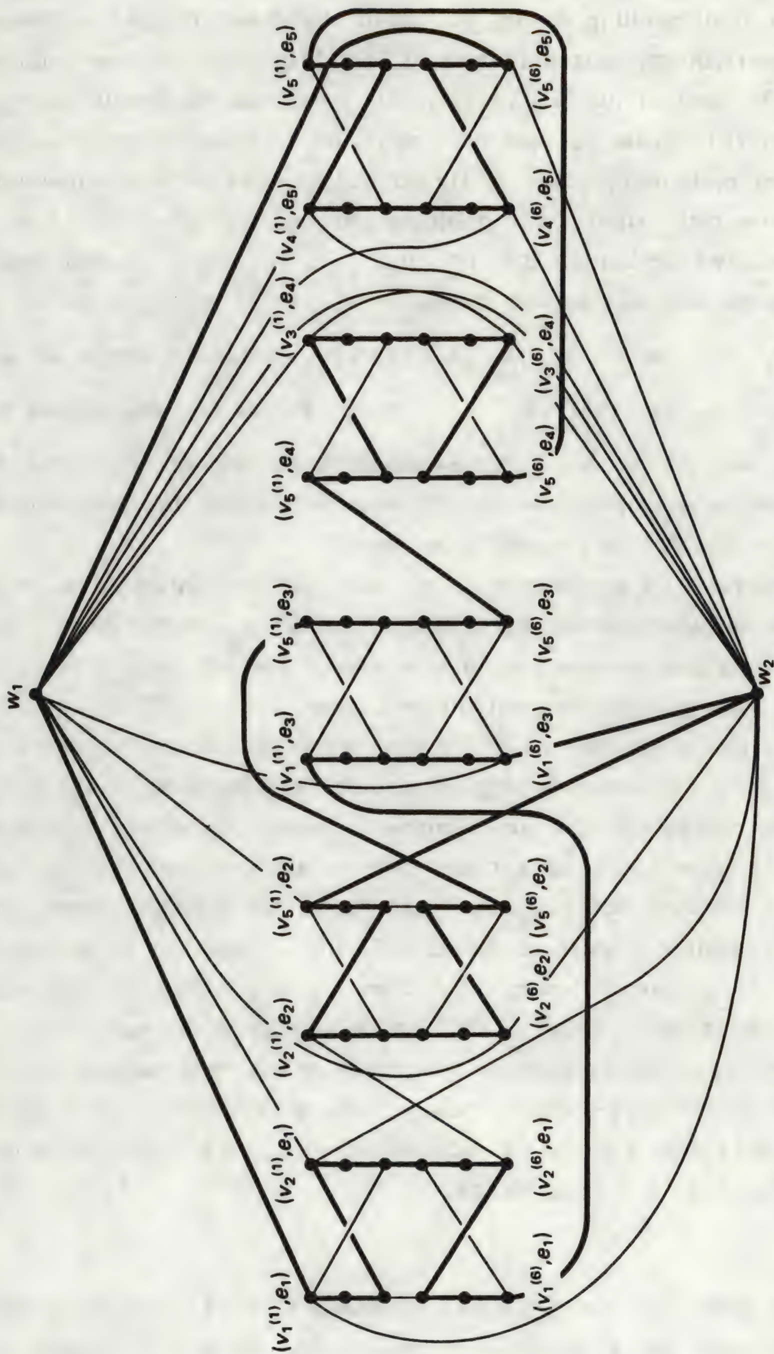
Figuur 6.7 De kanten die in een Hamiltoncircuit voorkomen

Als nu voor elke $v \in V$, $d = graad(v)$ dan is v het eindpunt van d kanten. Stel dat $e_{v[1]}, e_{v[2]}, \ldots, e_{v[d]}$ een aftelling van die kanten voorstelt. We verbinden nu de subgrafen die bij deze kanten behoren onderling door de volgende kanten in te voeren:
$(v^{(6)}, e_{v[1]})$——$(v^{(1)}, e_{v[2]})$, $(v^{(1)}, e_{v[2]})$——$(v^{(1)}, e_{v[3]})$, ..., $(v^{(6)}, e_{b[d-1]})$——$(v^{(1)}, e_{v[d]})$. Deze verzameling kanten, die we E'_v zullen noemen, zorgt ervoor dat er voor elke $v \in V$ in G' één enkel pad bestaat door alle knopen $(v^{(k)}, e)$ met $k = 1,2,\ldots,6$ en $e \in E$.

We voltooien nu de constructie van G' door de eerste en laatste knopen van elk van deze paden te verbinden met elk van de knopen $w_1, w_2, \ldots, w_b$. In figuur 6.9 is zo'n graaf G' getekend, die werd geconstrueerd uit de graaf G in figuur 6.8 met $b = 2$. Duidelijk is weer dat G' uit G in polynomiale tijd kan worden geconstrueerd en we beweren dat G' een Hamiltoncircuit bevat desd als G een knopenoverdekking met omvang b bezit.



Figuur 6.8 De graaf G



Figuur 6.9 De graaf G'

Stel dat V' een knopenoverdekking van G is met $\#(V') = b$. Deze laatste veronderstelling mogen we maken omdat we altijd extra knopen aan de overdekking mogen toevoegen; het blijft dan een overdekking. Stel dat V' bestaat uit $v_{\lambda 1}, v_{\lambda 2}, \ldots, v_{\lambda 6}$ en noem de graad van $v_{\lambda i}$, $d_{\lambda i}$ $(1 \leq i \leq b)$. Neem nu voor elke kant $e \in E$ tussen u en v kanten van G'_e op zoals aangegeven in figuur 6.7(a), (b) of (c), afhankelijk van de vraag of $\{u,v\} \in V'$ gelijk is aan $\{u\}$, $\{u,v\}$ of $\{v\}$. Vervolgens nemen we de kanten uit $E'_{v_{\lambda 1}} \cup E'_{v_{\lambda 2}} \cup \ldots \cup E'_{v_{\lambda k}}$ in het circuit op en verder ook alle kanten tussen w_i en $(v^{(1)}_{\lambda i}, e_{v_{\lambda i}[1]})$ $(1 \leq i \leq b)$ tussen w_{i+1} en $(v^{(6)}_{\lambda i}, e_{v_{\lambda i}[d_{\lambda i}]})$ $(1 \leq i \leq b)$. Tenslotte nemen we de kant tussen w_1 en $(v^{(6)}_{\lambda b}, e_{v_{\lambda b}[d_{\lambda b}]})$ op. Het is niet moeilijk na te gaan dat deze kanten altijd een Hamiltoncircuit vormen. In figuur 6.9 hebben we ze aangegeven onder de veronderstelling dat een overdekking met omvang 2 de knopen v_1, v_5 bevat.

Veronderstel nu omgekeerd dat G' een Hamiltoncircuit bevat en beschouw het gedeelte dat bij w_i begint en bij w_j eindigt voor $1 \leq i$, $j \leq b$, zodanig dat geen enkele andere knoop van het type w in het pad voorkomt. De eerste knoop op dit pad na w_i moet $(v^{(1)}, e)$ of $(v^{(6)}, e)$ zijn voor een $v \in V$ en een $e \in E$. Bekijk nu nog eens figuur 6.7 en de manier waarop we daarin de subgrafen, die met kanten uit de oorspronkelijke graaf corresponderen, onderling verbonden. We zien dat elke knoop op het pad naar w_j als tweede component een kant moet bevatten die v als eindpunt heeft. Het Hamiltoncircuit kan nu in b gedeelten worden onderverdeeld die bestaan uit de paden van $w_{\mu 1}$ naar $w_{\mu 2}$, van $w_{\mu 2}$ naar $w_{\mu 3}, \ldots$ en van $w_{\mu b}$ naar $w_{\mu 1}$ voor een of andere permutatie $\mu 1, \mu 2, \ldots, \mu b$ van de w-knopen. In geen van deze gedeelten liggen tussenliggende w-knopen en elk deel bepaalt dus een $v \in V$. Te zamen moeten deze b knopen een overdekking van V vormen omdat immers elke kant $e \in E$ uitkomt op een $v \in V$. Hiermee is het bewijs van stelling 6.10 geleverd.

Nu we weten dat het beslissingsprobleem TSP NP-compleet is moeten we concluderen dat we het niet in polynomiale tijd zullen kunnen oplossen, tenzij $P = NP$. We zijn er vrij zeker van dat $P \neq NP$ en dus zullen er vrij zeker gevallen van TSP zijn die niet in polynomiale tijd

kunnen worden opgelost. Ditzelfde geldt voor het oorspronkelijke handelsreizigersprobleem dat als een optimaliseringsprobleem werd geformuleerd, want anders zouden we het beslissingsprobleem kunnen oplossen door de kortst mogelijke route te bepalen en vervolgens te kijken of die kleiner is dan b. Willen we grote handelsreizigersproblemen in de praktijk oplossen dan hebben we een of andere heuristiek nodig. Een heuristiek is een soort vuistregel waarvan men intuïtief aanvoelt dat hij tot een aanvaardbare oplossing zal leiden. Een heel eenvoudige heuristiek voor het handelsreizigersprobleem is bijvoorbeeld: bezoek altijd als volgende stad de dichtstbijzijnde nog niet bezochte stad. We zullen deze heuristiek de 'naaste-buurman-heuristiek' noemen.

Iemand die heuristieken ontwerpt zal zoeken naar een methode die altijd oplossingen vindt in polynomiale tijd, waarbij de afwijking van de optimale oplossing binnen een aanvaardbare marge blijft. Als H een heuristiek is en $H(I)$ een oplossing volgens deze heuristiek voor een instantie I van een probleem Π, als verder $\mathrm{OPT}(I)$ de optimale oplossing voorstelt dan zou voor een minimalisatieprobleem moeten gelden:

$H(I) \leqslant \alpha \mathrm{OPT}(I) + \beta$ waarbij α en β constanten voorstellen.

Een goede heuristiek heeft een α in de buurt van 1 en een β in de buurt van 0. Als Π het handelsreizigersprobleem voorstelt en NN is naaste-buurman (Engels: Nearest Neighbour) heuristiek dan kan men aantonen, onder veronderstelling van de driehoeksongelijkheid $d_{ij} \leqslant d_{ik} + d_{kj}$ voor alle $1 \leqslant i, j, k \leqslant n$, dat

$$\mathrm{NN}(I) \leqslant \tfrac{1}{2}(\lceil \log_2 n \rceil + 1)\mathrm{OPT}(I)$$

als n het aantal steden is. Voor willekeurig grote n bestaan er nu problemen met

$$\mathrm{NN}(I) > \tfrac{1}{3}(\log_2(n+1) + \tfrac{4}{3})\mathrm{OPT}(I)$$

Al te best is de naaste-buurman methode dus niet!

De *prestatieverhouding* van een heuristiek H toegepast op een probleeminstantie I van een minimaliseringsprobleem Π, is gedefinieerd als

$$R_H(I) = \frac{H(I)}{\mathrm{OPT}(I)}$$

en de *absolute prestatieverhouding*, r_H van H voor Π is

$$R_H = \inf\{r \geqslant 1 \mid R_H(I) \leqslant r \text{ voor alle instanties } I \text{ van } \Pi\}$$

De *asymptotische prestatieverhouding*, R_H^∞ van H voor Π is

$$R_H^\infty = \inf\{r \geqslant 1 \mid \text{voor een } b \in Z^+,\ R_H(I) \leqslant r \text{ voor alle instanties } I \text{ van } \Pi \text{ die voldoen aan } \mathrm{OPT}(I) \geqslant b\}$$

Zowel de absolute als de asymptotische prestatieverhouding wordt gebruikt voor de analyse van de prestaties van heuristieke algoritmen. Hoe dichter bij 1 deze verhoudingen liggen des te beter is het algoritme. Voor NN geldt $R_{\mathrm{NN}} = R_{\mathrm{NN}}^\infty = \infty$; het gaat hier dus om een sterk af te raden heuristiek. Gelukkig bestaan er betere heuristieken voor het handelsreizigersprobleem, maar als niet is voldaan aan de driehoeksongelijkheid dan is een goede heuristiek niet waarschijnlijk omdat is aangetoond dat uit de veronderstelling $P \neq NP$ volgt dat er geen heuristiek H is die het handelsreizigersprobleem in polynomiale tijd oplost met een asymptotische prestatieverhouding $R_H^\infty < \infty$. Het is daarom verstandiger als we ons concentreren op het ontwerp van algoritmen die handelsreizigersproblemen meestal bevredigend oplossen, maar die een kleine kans hebben op een zeer slecht resultaat.

Voor een algemenere verhandeling over het ontwerp en de analyse van heuristieke algoritmen verwijzen we de lezer naar hoofdstuk 12 van Horowitz & Sahni (1978). Zie voor onderzoeksresultaten met betrekking tot het handelsreizigersprobleem het verzamelwerk *The Travelling Salesman Problem* (Lawler et al., 1984).

NUMERIEKE PROBLEMEN EN PSEUDO-POLYNOMIALE ALGORITMEN

In deze paragraaf bekijken we opnieuw de restricties die kunnen worden geformuleerd ten aanzien van coderingen van probleemvoorkomens. We zorgden er steeds nadrukkelijk voor dat de coderingen van getallen

die in een bepaald probleem voorkwamen werden uitgevoerd in de binaire notatie (of octaal, of decimaal, enzovoort, maar *niet* unair). Laten we nu eens beslissingsproblemen bekijken die geheel of gedeeltelijk worden beschreven door getallen die alle mogelijke geheeltallige waarden kunnen aannemen. Dergelijke problemen noemen wij *numerieke problemen*. TSP bijvoorbeeld valt in deze categorie omdat we geen beperkingen hebben opgelegd aan de lengte van de afstanden tussen de steden. Als $m \in Z$ een deel is van een instantie I van een numeriek probleem Π dan zorgen de restricties die we hebben opgelegd aan de codering e ervoor dat de substring van $e(I)$ die m codeert een lengte $0(\log m)$ heeft.

Alle definities in dit hoofdstuk hebben we geformuleerd in termen van lengte($e(I)$). Als we ons maar houden aan de restricties die we aan coderingen oplegden, dan doet het er niet echt toe welke codering we gebruiken. Als e en e' twee geschikte, beknopte coderingen zijn dan zijn lengte($e(I)$) en lengte($e'(I)$) *polynomiaal gerelateerd*, dat wil zeggen er bestaan polynomen p en p' zodanig dat

$$\text{lengte}(e(I)) \leqslant p(\text{lengte}(e'(I))$$

en $$\text{lengte}(e'(I)) \leqslant p'(\text{lengte}(e(I))$$

voor iedere $I \in D_\Pi$. Als lengte(I) voor een codering e, lengte($e(I)$) voorstelt dan weten we dat $\Pi \in P$ dan en slechts dan geldt als er een algoritme bestaat dat de oplossing vindt van elk voorkomen $I \in D_\Pi$ met een tijdcomplexiteit van de orde $0(\text{lengte}^k(I))$ voor een of andere constante $k \geqslant 0$. Komen er echter getallen in het probleem voor en laten we ook unaire coderingen toe dan is deze bewering niet langer waar. Als we een getal m in binaire notatie weergeven dan is daarvoor een string met lengte $\log_2(m) + 1$ nodig, maar in unaire notatie is de stringlengte m. De functies m en $\log_2(m)$ zijn niet polynomiaal gerelateerd: een functie die geen constante is en die polynomiaal in m is, is exponentieel in $\log_2(m)$.

Stel dat Π een willekeurig beslissingsprobleem is en geef het grootste getal dat in alle gegeven instanties I van Π voorkomt aan met max(I). Komen er geen getallen in I voor, geef max(I) dan de waarde 0. We definiëren een *numeriek probleem* nu formeel als ieder beslissingsprobleem Π zodanig dat er *geen* polynomiale functie p bestaat met

$\max(I) \in p(\text{lengte}(I))$ voor alle $I \in D_\Pi$.

Een algoritme van pseudo-polynomiale tijd voor Π is een algoritme dat Π oplost en dat een tijdcomplexiteit heeft die van boven wordt begrensd door een polynomiale functie van de twee variabelen lengte(I) en max(I). Alle algoritmen van polynomiale tijd zijn dus algoritmen van pseudo-polynomiale tijd omdat hun tijdfuncties immers van boven worden begrensd door een polynomiale functie van precies lengte(I). Het ziet ernaar uit dat het omgekeerde niet waar is en dat we dus misschien algoritmen van pseudo-polynomiale tijd kunnen vinden die NP-complete problemen oplossen. Dit is inderdaad het geval: een bekend voorbeeld is

Partitie

Gegeven: Een verzameling $A = \{a_1, a_2, \ldots, a_n\}$ van n elementen en een wegingsfunctie $w: A \to Z^+$.

Gevraagd: Kunnen we A partitioneren in twee deelverzamelingen van gelijk gewicht. Dat wil zeggen bestaat er een $A' \subseteq A$ zodanig dat

$$\sum_{a \in A'} w(a) = \sum_{a \in A \setminus A'} w(a)?$$

Het pseudo-polynomiale algoritme dat dit probleem oplost werkt als volgt. Noem som $= \sum_{a \in A} w(a)$. Als de waarde som niet even is dan is duidelijk dat het antwoord nee is: noem anders h = som/2. Construeer nu een matrix van Boole'se waarden, $M[i,j]$ met $1 \leqslant i \leqslant n$ en $0 \leqslant j \leqslant h$, zodanig dat $M[i,j] = T$ dan en slechts dan als er een deelverzameling van $\{a_1, a_2, \ldots, a_i\}$ bestaat met een gewicht van precies j. Deze waarden kunnen nu rij voor rij in de matrix m worden geplaatst. Merk ten eerste op dat $M[1,j] = T$ desd als ofwel $j=0$ ofwel $j = w(a_1)$. Elke volgende rij wordt nu berekend door gebruik te maken van de waarden uit de vorige rij en van het feit dat voor $1 < i \leqslant n$, $0 \leqslant j \leqslant h$, $M[i,j] = T$ desd als ofwel $M[i-1,j] = T$ ofwel $j \geqslant w(a_i)$ en $M[i-1, j-w(a_i)] = T$. Wanneer de hele matrix M op deze wijze is gevuld dan kan het partitieprobleem worden opgelost: het antwoord is 'ja' dan en slechts dan als $M[n,h] = T$. Eenvoudig kan worden aangetoond dat de tijdcomplexiteit van dit algoritme polynomiaal in nh is (zie oefening 6.14). Omdat het

partitieprobleem NP-compleet is zullen we geen algoritme kunnen vinden met een tijdcomplexiteitsfunctie die polynomiaal is in $n \log_2 h$.

Als gegeven is een beslissingprobleem Π en een polynoom p over de gehele getallen, dan definiëren we Π_p als het deelprobleem van Π dat die instanties I van Π bevat waarvoor geldt dat $\max(I) \leqslant p(\text{lengte}(I))$. We noemen een beslissingsprobleem Π *NP-compleet in strenge zin* dan en slechts dan als (i) $\Pi \in NP$ en (ii) er een polynoom p bestaat zodanig dat $\Pi_p \in NPC$.

Stelling 6.11

(1) Uit Π is NP-compleet en Π is geen numeriek probleem volgt dat Π NP-compleet is in strenge zin.

(2) Uit Π is NP-compleet in strenge zin volgt dat Π niet oplosbaar is met behulp van een pseudopolynomiaal algoritme, tenzij geldt dat $P = NP$.

Bewijs. (1) volgt onmiddellijk uit de definities.
(2) Stel dat $\Pi \in NP$ en dat er een p bestaat zodanig dat $\Pi_p \in NPC$. We zullen aantonen dat $\Pi_p \in P$ als Π oplosbaar is door middel van een algoritme van polynomiale tijd. Ons algoritme neemt een willekeurige inputstring x en gaat na of x een instantie I van Π codeert zodanig dat $\max(I) \leqslant p(\text{lengte}(I))$. We mogen veronderstellen dat de functies max en lengte in polynomiale tijd kunnen worden berekend en dus dat de ongelijkheid in polynomiale tijd kan worden gecontroleerd. Is het antwoord 'ja' dan passen we ons algoritme van pseudopolynomiale tijd voor Π op I toe. Omdat $\max(I) \leqslant p(\text{lengte}(I))$ is dit pseudopolynomiale algoritme polynomiaal in $\text{lengte}(I)$. Dus geldt $\Pi_p \in P$. Maar ook geldt $\Pi_p \in NPC$ en hieruit volgt het te bewijzen.

Uit deze stelling volgt onmiddellijk dat het partitieprobleem niet NP-compleet is in strenge zin. Er bestaan echter wel numerieke problemen in NPC die NP-compleet zijn in strenge zin. Een voorbeeld is TSP. We bewezen dat TSP NP-compleet is door een polynomiale transformatie $\text{HC} \propto \text{TSP}$ te beschrijven, maar de constructie was zodanig dat in elk

voorkomen van TSP al afstanden ter lengte 1 of 2 voorkomen. Hieruit blijkt dat een restrictie van TSP tot gevallen waarin alle afstanden $\leqslant 2$ zijn nog steeds *NP*-compleet is en dus is TSP *NP*-compleet in strenge zin.

AANVULLENDE PROBLEMEN

Als Π een beslissingsprobleem is dan is de vraagstelling van de algemene vorm: 'is, gegeven een instantie I. een bepaalde voorwaarde B waar voor I?'. Het *complement* van Π, meestal aangegeven door Π^c, is identiek aan Π, behalve dan dat de gestelde vraag is of de voorwaarde B onwaar is voor I. Π^c heeft dus als domein D_Π en $Y_{\Pi^c} = D_\Pi \setminus Y_\Pi$. Zo is bijvoorbeeld het complement van:

Samengesteld getal
Gegeven: Een positief geheel getal k.
Gevraagd: Bestaan er gehele getallen m en n zodat $k=mn$?

Priem
Gegeven: Een positief geheel getal k.
Gevraagd: Bestaan er geen gehele getallen m en $n>1$ zodat $k=mn$. Dat wil zeggen is k priem?

Als $\Pi \in P$ dan is duidelijk $\Pi^c \in P$. Gegeven een deterministische Turingmachine M die Π oplost, kunnen we gemakkelijk een DTM, M^c construeren die Π^c oplost, (zie oefening 6.15). M accepteert dus x desd als M^c x afwijst. De constructie die in de opgave wordt beschreven werkt omdat M deterministisch is en dus na elke input stopt. Het bewijs kan niet worden uitgebreid naar nondeterministische algoritmen.

Definiëren we $\text{co-}P = \{\Pi^c | \Pi \in P\}$ of op taalniveau $\text{co-}P = \{T^* \setminus L | L$ is een taal over T en $L \in P\}$ dan hebben we bewezen:

Stelling 6.12

co-P = P.

Laten we nu eens een NP-compleet probleem nemen, bijvoorbeeld TSP. Het complement van TSP is TSP^c.

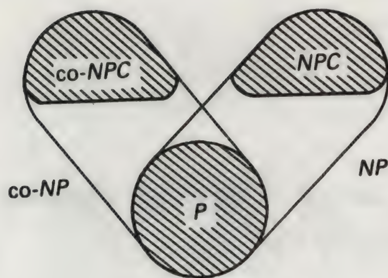
TSP^c

Gegeven: Een verzameling $C = \{c_1, c_2, \ldots, c_n\}$ van n steden met voor elk paar $c_i, c_j \in C$ een afstand $d_{ij} \in Z^+$ en een grens $b \in Z^+$.
Gevraagd: Is het onmogelijk voor een handelsreiziger in stad c_1 om alle steden in $\{c_2, \ldots, c_n\}$ precies een keer te bezoeken en terug te keren in c_1 met een totaal afgelegde afstand $\leqslant b$, dat wil zeggen kosten *alle* rondreizen meer dan b?

Het ziet er niet naar uit dat er een gemakkelijke manier bestaat om te bepalen of er 'ja' kan worden gezegd op TSP^c, zelfs niet door middel van intelligent raden. Er zit niets anders op dan alle rondreizen, of tenminste een zeer groot aantal, te onderzoeken. Het is dus niet waarschijnlijk dat TSP^c in polynomiale tijd kan worden opgelost, zelfs niet met een nondeterministische Turingmachine.

Als we dus definiëren co-P = $\{\Pi^c | \Pi \in NP\}$ of op taalniveau co-P = $\{T^* | L \backslash L$ is een taal over T en $L \in NP\}$ dan kunnen we op grond van het voorgaande veronderstellen dat $NP \neq$ co-NP. Eerder stelden we dat $NP \neq P$ en op die veronderstelling was onze hele theorie gebaseerd. Deze nieuwe veronderstelling is in zekere zin nog sterker want $NP \neq$ co-NP en $P = NP$ kan niet tegelijkertijd gelden. Dit zou leiden tot een onmiddellijke tegenspraak met stelling 6.12. Misschien zou wel kunnen gelden dat $P \neq NP$, hoewel NP = co-NP, maar erg waarschijnlijk is dit niet.

Als we nu co-NPC definiëren, dan volgen uit de volgende stelling een paar belangrijke eigenschappen. Zij zijn schematisch weergegeven in figuur 6.10 onder de veronderstelling dat $P \neq NP$ en $NP \neq$ co-NP beide gelden.



Figuur 6.10

Stelling 6.13

(1) $P \subseteq (NP \cap \text{co-}NP)$.

(2) Uit $NP \neq \text{co-}NP$ volgt $NPC \subseteq (NP \backslash \text{co-}NP)$ en ook $\text{co-}NPC \subseteq (\text{co-}NP \backslash NP)$.

Bewijs

(1) $P \subseteq NP$ en $P = \text{co-}P \subseteq \text{co-}NP$ en hieruit volgt het te bewijzen.

(2) We zullen bewijzen dat $NPC \subseteq (NP \backslash \text{co-}NP)$. Het tweede deel van het bewijs volgt dan uit symmetrie-overwegingen. Per definitie geldt $NPC \subseteq NP$ en we moeten dus aantonen dat er geen enkel probleem in NPC voorkomt dat ook in co-NP ligt. Laten we eens veronderstellen dat er wel een dergelijk probleem bestaat en noem het Π. We zullen zien dat deze veronderstelling tot een logische tegenspraak leidt. Als Π_1 een willekeurig probleem is in NP dan geldt $\Pi_1 \propto \Pi$ omdat $\Pi \in NPC$. Het is dan eenvoudig aan te tonen dat ook $\Pi_1^c \propto \Pi^c$. Omdat nu $\Pi \in \text{co-}NP$ en $\Pi^c \in NP$ en omdat $\Pi_1^c \propto \Pi^c$ moet gelden dat $\Pi_1^c \in NP$. We hebben nu dus bewezen dat $\text{co-}NP \subseteq NP$. Dus is $NP = \text{co-}(\text{co-}NP) \subseteq \text{co-}NP$ en dus is $NP = \text{co-}NP$ en dit is de gezochte tegenspraak.

We zagen dat het probleem *Samengesteld getal* in NP ligt en we stelden dat waarschijnlijk is dat het in $NPI = NP \backslash (NPC \cup P)$ ligt. Het complement van dit probleem *Priem* ligt ook in NP en dit sterkt ons in het vermoeden dat *Samengesteld getal* niet in NPC ligt. Als dat wel zo was dan hadden we immers bewezen dat $NP = \text{co-}NP$. Natuurlijk is het nog steeds mogelijk dat *Samengesteld getal* in P ligt. Men blijft zoeken naar

een bewijs of een bewijs van het tegendeel. Omdat verscheidene coderingssystemen voor het beveiligen van gegevens berusten op het in priemfactoren ontbinden van getallen is dit onderzoek niet zonder zin.

DE POLYNOMIALE HIERARCHIE

Stel dat Π een beslissingsprobleem is en stel dat we een methode hebben ontwikkeld om het op te lossen. Iemand die deze methode gebruikt hoeft alleen maar een instantie $I \in D_{\Pi'}$ van het probleem als input te geven en als resultaat volgt een 0 of een 1, afhankelijk van het antwoord op de vraag of $I \in Y_{\Pi'}$. Het hoeft de gebruiker van de methode niet te interesseren hoe deze werkt, zolang maar altijd het juiste resultaat wordt opgeleverd. Laten we nu verder veronderstellen dat het toepassen van de methode altijd maar een enkele tijdseenheid kost, hoe moeilijk Π' ook is en hoe veel tijd het ook kost om het probleem op te lossen. Zo'n methode zullen we een *Orakel* noemen.

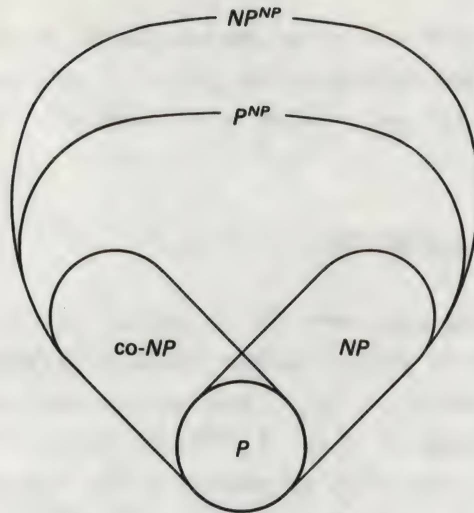
Een probleem Π heet *NP-gemakkelijk* als het in polynomiale tijd kan worden opgelost door een deterministisch algoritme dat gebruik kan maken van een orakel dat een of ander probleem in NP oplost. Elk probleem in NP is NP-gemakkelijk - we hoeven onszelf maar van een orakel te voorzien om het op te lossen. Als $\Pi \in NP$ dan kan een orakel dat Π oplost ook worden gebruikt voor het ontwerpen van een triviaal algoritme voor het oplossen van Π^c en dus is elk probleem in co-NP ook NP-gemakkelijk.

Als X een willekeurige klasse van problemen is dan definiëren we

$$P^X = \{\Pi \mid \Pi \text{ kan in polynomiale tijd worden opgelost door een deterministisch algoritme dat toegang heeft tot een orakel dat een probleem } \Pi' \in X \text{ oplost}\}.$$

De NP-gemakkelijke problemen zijn dan juist de problemen in P^{NP}. Eerder toonden we aan dat $P^{NP} \supseteq NP \cup \text{co-}NP$ en onder de voorwaarde dat $NP \neq \text{co-}NP$ is hier sprake van een echte deelverzameling.

Na de definitie van P^X is een logische volgende stap de definitie van NP^X.



Figuur 6.11

NP^X = {$\Pi | \Pi$ kan in polynomiale tijd worden opgelost door een nondeterministisch algoritme dat toegang heeft tot een orakel dat een probleem $\Pi' \in X$ oplost}.

Dan geldt $NP^{NP} \supseteq P^{NP} \supseteq NP \cup \text{co-}NP$ zoals in figuur 6.11 schematisch is weergegeven. We vermeldden al dat het waarschijnlijk is dat de tweede deelverzameling echt is en er zijn goede redenen om dit ook van de eerste te veronderstellen.

Behalve NP-gemakkelijke problemen kennen we ook NP-moeilijke problemen. Een probleem Π is NP-moeilijk als er een NP-compleet probleem Π' bestaat dat zou kunnen worden opgelost door een deterministisch algoritme van polynomiale tijd als dit de beschikking heeft over een orakel dat Π oplost. Uit deze definitie volgt onmiddellijk dat een beslissingsprobleem niet NP-moeilijk kan zijn en tegelijkertijd in P kan liggen, tenzij $P = NP$. Een voorbeeld van een probleem dat NP-moeilijk is en niet NP-gemakkelijk lijkt te zijn is het volgende.

Minimale equivalente expressie (MEE)

Gegeven: Een correcte logische expressie E, waarin literalen voorkomen over een eindige verzameling van variabelen, constanten, de waarden T (true=waar) en F (false=onwaar), de logische operatoren $\wedge$ (en), $\vee$ (of), $\sim$ (niet) en $\Rightarrow$ (impliceert) en een $k \in Z^+$.

Gevraagd: Bestaat er een correcte logische expressie E' die $\leqslant k$ literalen bevat en die logisch equivalent is met E?

Het bewijs dat MEE NP-moeilijk is laten we als oefening aan de lezer over (zie oefening 6.16).

Hoewel niemand nog heeft kunnen aantonen dat MEE in P^{NP} is en hoewel algemeen wordt verondersteld dat dit niet zo is, komt het wel in NP^{NP} voor. We bewijzen dit door het orakel een methode te geven die SAT oplost. Het nondeterministische algoritme dat MEE oplost raadt dan een expressie E' met $\leqslant k$ literalen en gebruikt vervolgens het orakel om na te gaan of aan $\sim((E \Rightarrow E') \wedge (E' \Rightarrow E))$ voldaan kan worden. Als dat niet zo is dan moet het gegeven probleem in Y_{MEE} liggen.

De hiërarchie uit figuur 6.9 is als volgt gegeneraliseerd naar een niet eindige hiërarchie.

$$\Delta_0^p = \Pi_0^p = \Delta_0^p = P$$

en voor alle $k \leqslant 0$

$$\Sigma_{k+1}^p = P^{\Sigma_k^p}$$

$$\Sigma_{k+1}^p = NP^{\Sigma_k^p}$$

en $\Pi_{k+1}^p = \text{co-}\Sigma_{k+1}^p$

Dus geldt:

$$\Delta_1^p = P^p = P, \quad \Sigma_1^p = NP^p = NP, \quad \Pi_1^p = \text{co-}NP$$

$$\Delta_2^p = P^{NP}, \quad \Sigma_2^p = NP^{NP}, \quad \Pi_2^p = \text{co-}NP^{NP}, \text{ enzovoort.}$$

Men kan aantonen dat voor elke k $\Delta_k^p \subseteq \Pi_k^p \cap \Sigma_k^p$ en $\Pi_k^p \cup \Sigma_k^p \subseteq \Pi_{k+1}^p$ zoals weergegeven in figuur 6.12. Deze hele hiërarchie van complexiteitsklassen wordt de *polynomiale hiërarchie* genoemd. De verzameling $PH = \bigcup_{k=1}^{\infty} \Sigma_k^p$ vertegenwoordigt alle problemen uit deze hiërarchie. Zie

Stockmeyer (1976) voor een verdere behandeling. In Garey & Johnson (1978) wordt de orakel Turingmachine behandeld en worden de ideeën waarvan we hier een overzicht gaven verder uitgewerkt en geformuleerd.

Δ^p_{k+1}

Π^p_k Σ^p_k

Δ^p_k

Figuur 6.12

RUIMTEBEPERKINGEN

Tot dusverre hielden we ons in dit hoofdstuk alleen bezig met beperkingen ten aanzien van de beschikbare rekentijd. Ook computergeheugen is echter beperkt beschikbaar en in praktische situaties kan deze beperking zich wel eens sterker doen gelden dan de beperking in rekentijd. We noemen de hoeveelheid computergeheugen die nodig is om een algoritme uit te voeren de *ruimtebehoefte* van dit algoritme. Voor een Turingmachine is de ruimtebehoefte eenvoudig het aantal verschillende locaties op de tape die door de lees/schrijfkop zijn gelezen of beschreven. Het aantal verschillende bezochte locaties kan nooit groter zijn dan de rekentijd die voor het algoritme nodig was en dus moet elk algoritme dat in polynomiale tijd oplosbaar is ook in polynomiale ruimte oplosbaar zijn. We definiëren daarom:

PRUIMTE = {*L*| er bestaat een deterministische Turingmachine met polynomiaal begrensde ruimte die na alle inputs halt houdt en die de taal *L* accepteert}

of in termen van problemen

$PRUIMTE = \{\Pi | \Pi$ is een beslissingsprobleem dat kan worden opgelost door een DTM met polynomiaal begrensde ruimte die na alle inputs halt houdt$\}$

Er geldt $P \subseteq PRUIMTE$ en uit het bewijs van stelling 4.8 volgt dat ook $NP \subseteq PRUIMTE$. Ook geldt co-$NP \subseteq PRUIMTE$ en ook de hele polynomiale hiërarchie $PH \subseteq PRUIMTE$. *PRUIMTE* is dus een grote klasse van problemen.

Hoewel alle problemen die oplosbaar zijn in polynomiale tijd kunnen worden opgelost in een polynomiale ruimte, is het niet bekend of er problemen bestaan die oplosbaar zijn in polynomiale ruimte, maar die niet oplosbaar zijn in polynomiale tijd. Toch lijkt dit een waarschijnlijke veronderstelling; als $P \neq NP$ of zelfs $P \neq PH$ dan volgt daaruit dat $P \neq PRUIMTE$.

Net als bij *NP* kunnen we ook in *PRUIMTE* een klasse problemen aangeven die in bepaalde zin als de moeilijkste kunnen worden beschouwd. Als we uitgaan van het taalmodel dan noemen we *L* *PRUIMTE*-compleet desd als $L \in PRUIMTE$ en als voor alle $L' \in PRUIMTE$, $L' \propto L$. Hieruit volgt dat als *L* *PRUIMTE*-compleet is dan geldt $L \in P$ desd als $P = PRUIMTE$. We kunnen deze definities nu op voor de hand liggende wijze uitbreiden zodat zij op problemen van toepassing zijn. Zelfs als zou gelden $P = NP$ dan nog zou kunnen gelden dat $P \neq PRUIMTE$ en *PRUIMTE*-compleetheid is daarom een sterkere aanduiding voor onhandelbaarheid van een probleem dan *NP*-compleetheid. In de praktijk hebben we echter meestal te maken met *NP*-complete problemen en dit is de reden dat we hierop in dit boek vooral ingaan.

Het kernprobleem dat *PRUIMTE*-compleet is en dat dezelfde rol speelt als SAT bij de *NP*-complete problemen is het volgende.

De Gekwantificeerde Boole'se Formule (QBF)
Gegeven: Een correct gevormde Boole'se formule

$$F = (Q_1x_1)(Q_2x_2)\dots(Q_nx_n)E$$

E is een Boole'se expressie over de variabelen $x_1,x_2,...,x_n$ en de grootheden Q_1 stellen ofwel de logische existentiekwantor ($\exists$) ofwel de logische universaliteitskwantor ($\forall$) voor.
Gevraagd: Is F waar?

Nu we *PRUIMTE* hebben gedefinieerd zou men kunnen verwachten dat vervolgens *NPRUIMTE* wordt gedefinieerd - de klasse van talen die worden herkend door nondeterministische Turingmachines met gebruik van een polynomiaal begrensde ruimte. Het geven van zo'n definitie is echter zinloos omdat werd aangetoond dat *PRUIMTE* = *NPRUIMTE*. We gaan dus niet zoeken naar complexiteitsklassen groter dan *PRUIMTE*, maar juist naar verdere inperkingen van die ruimte tot een kleinere complexiteitsklasse.

We gaan uit van een deterministische Turingmachine met twee tapes - - een inputtape waarvan alleen gelezen wordt en een lees/schrijf werktape waarop we uiteindelijk het resultaat zullen vinden. Bij het meten van de benodigde ruimte lijkt het zinnig alleen de ruimte op de werktape te meten. Zeker als we ruimterestricties beschouwen die sublineair zijn (dus $\Omega(n)$ maar niet $O(n)$) dan zijn we wel gedwongen dit te doen omdat voor de input in ieder geval n locaties nodig zijn. Recent onderzoek concentreerde zich op de klasse

DLOGRUIMTE = $\{L|L$ kan worden herkend door een deterministisch Turingmachineprogramma met een werkruimte begrensd door $\log_2 n+1$ als n de lengte van de inputstring is$\}$.

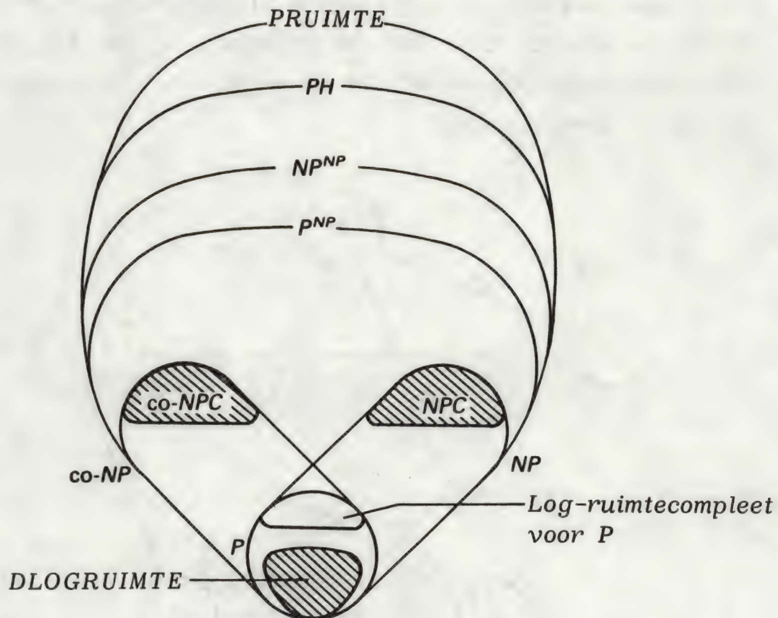
Aantonen dat *DLOGRUIMTE* $\subseteq P$ is betrekklijk eenvoudig maar niemand weet nog of hier sprake is van een echte deelverzameling. Men kan bewijzen dat P = *DLOGRUIMTE* en P = *PRUIMTE* niet tegelijkertijd waar kunnen zijn en men veronderstelt dat zelfs geen van beide waar is. We vermoeden dus dat er problemen in P voorkomen die een grotere dan logaritmische werkruimte nodig hebben voor hun oplossing.

Stel dat L_1 en L_2 talen zijn over de alfabetten T_1 en T_2. We definiëren dat een *log-ruimtetransformatie* van L_2 naar L_1 als een functie $f: T_1^* \to T_2^*$ zodanig dat

(1) $x \in L_1$ desd als $f(x) \in L_2$ en

(2) f kan worden berekend door een DTM met een werkruimte die wordt begrensd door $\lceil \log_2 x + 1 \rceil$ voor elke inputstring $x \in T^*$. Als er zo'n log-ruimtetransformatie van L_1 naar L_2 bestaat dan noteren we dat als $L_1 \propto_{\mathrm{LOG}} L_2$.

Men kan bewijzen dat $\propto_{\mathrm{LOG}}$ transitief is en dat als $L_1 \propto_{\mathrm{LOG}} L_2$ en $L_2 \in \mathit{DLOGRUIMTE}$ dat dan ook $L_1 \in \mathit{DLOGRUIMTE}$. We definiëren daarom een taal $L \in P$ als *log-ruimtecompleet* voor P als en alleen als voor alle $L' \in P$ geldt dat $L' \propto_{\mathrm{LOG}} L$. Als L log-ruimtecompleet is voor P dan geldt $L \in \mathit{DLOGRUIMTE}$ desd als $\mathit{DLOGRUIMTE} = P$. Deze probleemklasse is op dit moment onderwerp van veel studies; de veronderstelling is geuit dat problemen die log-ruimtecompleet zijn voor P hoogstwaarschijnlijk problemen zijn die niet beduidend sneller zullen kunnen worden opgelost met behulp van parallelle algoritmen. Dit vermoeden wordt de *Parallelle rekenhypothese* genoemd.



Figuur 6.13

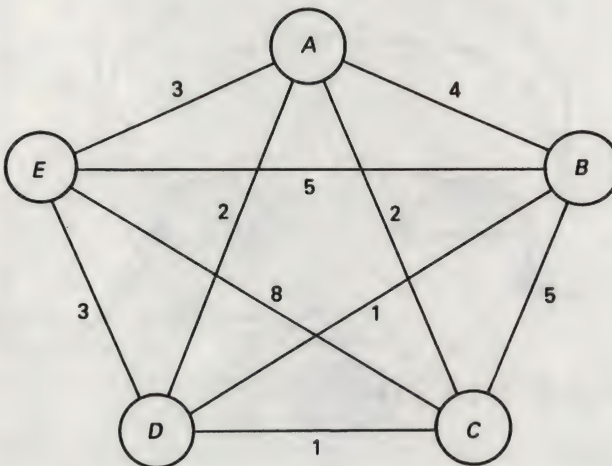
SAMENVATTING

Ons huidige zicht op de wereld der complexiteitsklassen is in figuur 6.13 schematisch weergegeven. Er zijn nog veel meer complexiteitsklassen voorgesteld en onderzocht, maar behandeling daarvan in dit boek zou te ver voeren. In ieder geval behandelden we de belangrijkste klassen en in het bijzonder die klassen die voortkomen uit praktijkproblemen. Hierdoor hebben wij een verband kunnen leggen tussen een interessant stuk formele theorie en de praktijk.

OEFENINGEN

1. Laat zien dat
   (a) 2^n is $O(n!)$;
   (b) $\log_2(n)$ is $\theta(\log_k(n)$ voor iedere $k \geqslant 2$;
   (c) $f(n)$ is $\theta(g(n))$ impliceert dat $g(n)$ is $\theta(f(n))$.

2. Bepaal een oplossing voor het handelsreizigersprobleem bij de gelabelde graaf uit figuur 6.14. Moeten alle 4! mogelijke rondreizen worden berekend? Als A het uitgangspunt is, zou NN dan de juiste oplossing hebben gegeven? Wat is het slechtste resultaat waartoe NN bij dit probleem leidt?



Figuur 6.14

3. A en B zijn twee $n \times n$ matrices met $n = 2^k$. A en B worden ieder onderverdeeld in vier $n/2 \times n/2$ deelmatrices, zoals hieronder aangegeven.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Als de produktmatrix $C = AB$ op dezelfde manier wordt onderverdeeld geef dan uitdrukkingen voor C_{11}, C_{12}, C_{21} en C_{22} in termen van

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

De waarden P,Q,R,S,T,U,V kunnen worden berekend met behulp van 7 matrixvermenigvuldigingen en 10 matrixoptellingen/aftrekkingen. Hoeveel extra optellingen/aftrekkingen zijn nodig om uit deze waarden C_{11}, C_{12}, C_{21} en C_{22} te berekenen? Als $T(n)$ de totale tijd voorstelt die nodig is om twee $n \times n$ matrices met elkaar te vermenigvuldigen leid dan de volgende relatie af

$$T(n) = \begin{cases} b & \text{voor } n \leqslant 2, \\ 7T(n/2) + an^2 & \text{voor } n > 2. \end{cases}$$

Hierbij stellen a en b constanten voor. Leid nu een $0(n^{\log_2 7})$ algoritme af voor de vermenigvuldiging van twee $n \times n$ matrices. (Dit algoritme staat bekend als het algoritme van Strassen.)

4. Toon aan dat P een equivalentieklasse is onder polynomiale equivalentie. [Hint: los het probleem via de transformatie op.]

5. Als M een NDTM is van graad m, $(m > 1)$, die een taal L accepteert dan bestaat er een NDTM, M' van graad 2 die L ook accepteert terwijl voor alle $x \in L$, $l_{M'}(x)$ $0((\log m)l_M(x))$ is. Bewijs dit.

6. Hoeveel Boole'se variabelen en hoeveel clausen worden er in het bewijs van de stelling van Cook geconstrueerd?

7. (a) Beschrijf een polynomiaal algoritme dat 2SAT oplost. [Hint: als aan $\{A, B\}$ en aan $\{\bar{B}, C\}$ kan worden voldaan dan geldt dit ook voor $\{A, C\}$.]
(b) Bewijs dat kSAT voor $k \geqslant 4$ NP-compleet is.

8. Beschouw het volgende probleem:

Exacte overdekking door 3-elementsverzamelingen (X3C)
Gegeven: Een verzameling X met $|X| = 3q$ en een collectie C van 3-elementsdeelverzamelingen van X.
Gevraagd: Bevat C een exacte overdekking voor X. Dat wil zeggen bestaat er een deelcollectie C' van C zodanig dat elk element van X voorkomt in een element van C'?

(a) Construeer een voorkomen $I \in Y_{\text{X3C}}$ en een voorkomen $I' \notin Y_{\text{X3C}}$.
(b) Toon aan dat $\text{X3C} \in NP$.
(c) Bewijs dat X3C NP-compleet is door middel van een polynomiale transformatie $\text{3DM} \propto \text{X3C}$.

9. Zij G een graaf met knopen V en kanten E. Zij $c: E \to Z^+$ een kostenfunctie gedefinieerd op de kanten van G. Zij verder $V' \subseteq V$. Beschouw nu het volgende probleem: te berekenen de subgraaf van G met minimale kosten, die verbonden is en die alle knopen uit V' (en mogelijk nog andere) bevat. Dit probleem uit de grafentheorie heet het Steinertree probleem.

(a) Bewijs dat de oplossing een boom moet zijn.
(b) Bewijs dat het probleem oplosbaar is in polynomiale tijd als $V' = V$ of als $\#(V') = 2$.
(c) Zij $\#(V) = n$ en $\#(V') = m$. Toon aan dat het probleem kan worden opgelost door 2^{n-m} keer een probleem van de minimale opspannende boom op te lossen als $2 < m < n$.
(d) Formuleer het beslissingsprobleem bij het Steinertree probleem voor grafen. Toon aan dat dit probleem NP-compleet is. [Hint: construeer een polynomiale transformatie vanuit X3C.]

10. G is een graaf met knopenverzameling V en kantenverzameling E. Een *kliek* in G is een deelverzameling $V' \subseteq V$ zodanig dat elk knopenpaar in V' door een kant uit E verbonden is. Het *complement* van G heeft knopen V en een kant ⓤ——ⓥ desd als u en v in G G niet onderling verbonden zijn. Laat zien dat V' een kliek is in het complement van G desd als $V \backslash V'$ een knopenoverdekking voor G vormt. Leid hieruit af dat het volgende probleem NP-compleet is.

    Kliek
    Gegeven: Een graaf G en een positief geheel getal b.
    Gevraagd: Bevat G een kliek van $\geqslant b$ knopen?

11. Toon aan dat het niet waarschijnlijk is dat we een algoritme van polynomiale tijd zullen vinden om het *subgraaf-isomorfisme-probleem* op te lossen. Dit probleem bestaat uit het bepalen van een subgraaf in een gegeven graaf die isomorf is met een andere gegeven graaf. [Hint: zie vraag 10.]

12. Bewijs dat het volgende probleem NP-compleet is.

    Multiprocessorscheduling
    Gegeven: Een eindige verzameling J van taken (jobs) en een duur $l(j) \in Z^+$ voor elke taak j. Verder $n \in Z^+$ processoren en een uiterst oplevertijdstip (deadline) $d \in Z^+$.
    Gevraagd: Bestaat er een partitie $J = J_1 \cup J_2 \cup \ldots \cup J_n$ van J in n disjuncte verzamelingen zodanig dat

    $$\max\left\{ \sum_{j \in J_i} l(j) \,\middle|\, 1 \leqslant i \leqslant m \right\} \leqslant d?$$

13. Een Hamiltonpad in een graaf is een pad dat elke knoop in de graaf precies één keer bezoekt. Toon, door het bewijs van stelling 6.10 aan te passen, aan dat het beantwoorden van de vraag of een graaf een Hamiltonpad heeft een NP-compleet probleem is.

14. Beschouw het partitieprobleem met 8 elementen $a_1, a_2, \ldots, a_8$ met gewichten 2,3,4,5,6,7,8. Gebruik het algoritme van pseudopolynomiale tijd dat we in dit hoofdstuk beschreven om na te gaan of er een partitie bestaat. Laat zien dat het algoritme $0(nh)$ is, waarbij n het aantal elementen voorstelt en $h = \Sigma a_i/2$.

15. Als $L \in P$ dan bestaat er een DTM, M die L accepteert en die een unieke acceptatietoestand (dit is een eindtoestand) en een unieke afwijstoestand (dit is een toestand waarin de machine zonder succes halthoudt) heeft. Verder geldt dan $L \in P$ desd $L^c \in P$. Bewijs deze beweringen en geef aan waarom uw constructie niet werkt voor $L \in NP$.

16. Bewijs dat een DTM met een orakel dat MEE oplost kan worden gebruikt voor het oplossen van SAT in polynomiale tijd. Leid hieruit af dat MEE een NP-moeilijk probleem is.

17. Bewijs dat elk NP-gemakkelijk probleem in $P^{\{SAT\}}$ ligt.

18. Bewijs dat $NP \in PRUIMTE$. [Hint: zie stelling 4.8.]

Appendix

De Turingmachine Simulator

We beschrijven hier een programma in Pascal dat kan worden gebruikt om het gedrag van een Turingmachine $M = (Q, \Sigma, T, P, q\ , F)$ te simuleren. Als enige beperking eisen we dat de gebruikte hoeveelheid tape eindig is. Zonder verlies van algemeenheid veronderstellen we dat de toestanden Q worden gevormd door de natuurlijke getallen $1, 2, \ldots, n$ voor een $n \geqslant 1$ en dat de eindtoestanden F worden gevormd door de toestanden $m, m+1, \ldots, n$ voor $1 \leqslant m \leqslant n$. De symbolen uit het tape alfabet Σ mogen lengte $\geqslant 1$ hebben, maar we eisen dat zij allen dezelfde lengte hebben. Hiertoe zal het misschien nodig zijn bepaalde symbolen aan te vullen met spaties. Als dus bijvoorbeeld $\Sigma = \{0,01,02\}$ dan is de lengte van ieder symbool in Σ gelijk aan twee en we vullen de symbolen dus aan tot {' 0','01','02'}.

We voeren een TM-programma in twee stappen in in de simulator. Allereerst moeten de volgende constanten worden geïnitialiseerd.

AantalToestanden: het aantal toestanden in Q. Vervolgens wordt aangenomen dat de toestanden worden gevormd door de natuurlijke getallen 1,1,...,AantalToestanden.

BeginToestand: het getal dat de begintoestand vertegenwoordigt.

EersteHaltToestand: het kleinste gehele getal dat een eindtoestand voorstelt. Vervolgens wordt aangenomen dat alle toestanden in het interval EersteHaltToestand..Aantal Toestanden een eindtoestand is.

AantalSymbolen: het aantal verschillende symbolen in het tape alfabet Σ.

MaxSymboolLengte: lengte van de langste symbolenstring in Σ. Vervolgend wordt aangenomen dat alle symbolen die lengte hebben.

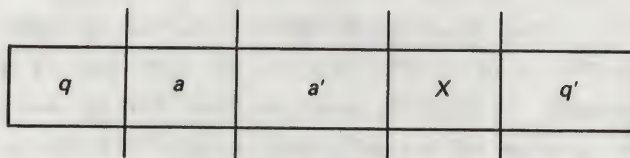
Blank: het symbool dat het lege symbool aangeeft (en dat natuurlijk de lengte MaxSymboolLengte moet hebben).

Spoor: een Boole'se variabele die TRUE is als we alle tussenresultaten willen zien.

Tapelimiet: de grootheid die de lengte van de tape bepaalt. De tape beschikt over de locaties –Tapelimiet..Tapelimiet.

Ten tweede vullen we een file TupelFile met het TM-programma P. Als $P(q,a) = (q',a',X)$, $q,q' \in Q$, $a' \in \Sigma$, $X \in \{L,R,0\}$ dan staat in de file een regel zoals is aangegeven in figuur A.1. Hierbij zijn

q,q' van het type TOESTANDTYPE =1..AantalToestanden
a,a' van het type
SYMBOOLTYPE = PACKED ARRAY[1..MaxSymboollengte] of CHAR,
en X heeft het type STAP = CHAR.



Figuur A.1

De simulator eist niet dat de bestandsregels in een bepaalde volgorde staan en ook het alfabet Σ hoeft niet van te voren bekend te zijn. Door het lezen van Tupelfile wordt de inhoud van Σ automatisch vastgesteld.

Nadat de Turingmachine M is beschreven aan de simulator, moet de input worden gegeven door middel van de file TMtape. TMtape is een file van het type Symbooltype en elk symbool moet precies MaxSymboollengte tekens bevatten. Het programma is geschreven in Turbo Pascal, versie 3.0. Ter vereenvoudiging gingen we uit van symbolen van één teken en is aan Symbooltype het type 'char' gegeven.

Voor het vullen van de Tupelfile met records van het type Vijftupel moet een afzonderlijk invoerprogramma worden geschreven. Van de inputtape TMtape maakten we een textfile; deze kan met een texteditor zoals Edlin worden gevuld.

Na de programmalisting volgt de output voor de binair-unair omzetter uit figuur 2.9. Als inputtape gebruikten we het binaire getal 101; de unaire output zou dus 11111 moeten worden en dit blijkt inderdaad het

geval. (Natuurlijk is dit verre van een correctheidsbewijs van het programma!)

We drukten ter wille van de ruimte alleen die stappen af waarbij iets anders op de tape wordt geschreven dan werd gelezen.

```
program TuringMachine;
const
  Dummy              =-1;
  Blanco             =0;
  AantalToestanden   =23;
  Begintoestand      =1;
  EersteHalttoestand =23;
  AantalSymbolen     =5;
  MaxSymboollengte   =1;
  Blank              ='^';
  Spoor              =true;
  Tapelimiet         =1000;

{Bovenstaande constanten moeten tevoren worden ingesteld             }
{De huidige waarden zijn voor de simulatie van de binair-unairconvertor}
{(Zie: figuur 2.9)}

type
  Symbooltype = char;
  Toestandtype = 1..AantalToestanden;
  Symcode     = 0..AantalSymbolen;
  Actie       = record
                  SchrijfSymbool : Symcode;
                  Stap           : char;
                  NieuweToestand : integer
                end;
  Vijftupel   = record
                  HuidigeToestand: integer;
                  LeesSymbool    : Symbooltype;
                  SchrijfSymbool : Symbooltype;
                  Stap           : char;
                  NieuweToestand : integer;
                end;
  Tapeindex   = -Tapelimiet..Tapelimiet;

var
  Alfabet        :array[Symcode] of Symbooltype;
  Overgang,
  GevrActie      :Actie;
  OvergangsTabel :array[Toestandtype,Symcode] of Actie;
  Tape           :array[Tapeindex] of Symcode;
```

```
  Symbool       :Symbooltype;
  Tupel         :Vijftupel;
  FunctieDef,
  TapeVol,
  EindTape      :boolean;
  SymBekend,
  Gelezen       :Symcode;
  Toestand      :Toestandtype;
  Symboolnr     :Symcode;
  Tapekop,
  Kop,
  MeestLinker,
  MeestRechter  :Tapeindex;
  TMtape        :{file of Symbooltype}text;
  TupelFile     :file of Vijftupel;
  Nstap         :integer;
  Ans           :string[1];
  Opt           :text;

function BepSymcode(Zoeksym:Symbooltype):Symcode;
{Geeft geheeltallige codering van symbool                      }
{Als symbool nog niet voorkomt wordt dit aan het eind toegevoegd}
{en wordt de nieuwe positie gegeven                             }
var
  Gevonden: boolean;
  Symnum  : Symcode;
begin
  Gevonden:=false;Symnum:=0;
  while(Symnum<=SymBekend) and not Gevonden do
    if Alfabet[Symnum]=Zoeksym then
      Gevonden:=true
    else
      Symnum:=Symnum+1;
  if Symnum>SymBekend then
  begin
    Alfabet[Symnum]:=Zoeksym;
    SymBekend:=SymBekend+1
  end;
  BepSymcode:=Symnum
end;

procedure ToonResultaat;
{Drukt de niet lege tekens van de tape af. Allemaal als het een tussen-}
{resultaat is en vanaf positie 1 als het om het eindresultaat gaat.    }
var
```

```
  Ptr,LPtr,RPtr: Tapeindex;
begin
  if Spoor and not EindTape then
  begin
    LPtr:=MeestLinker;
    while (LPtr<=0) and (Tape[LPtr]=Blanco) do
      LPtr:=LPtr+1;

    with GevrActie do
    begin
      if Stap='R' then
        Kop:=Tapekop-1
      else
        if Stap='L' then
          Kop:=Tapekop+1
        else Kop:=Tapekop;
       write(Opt,Nstap:4,Toestand:4,Kop:4,Alfabet[Gelezen]:4,
                                      Alfabet[Schrijfsymbool]:4,Stap:4);
    end
  end
  else
  begin
    LPtr:=1;
    writeln(Opt,'Berekende output is: ')
  end;
  RPtr:=MeestRechter;
  while (RPtr>=1) and (Tape[RPtr]=Blanco) do
    RPtr:=RPtr-1;
  for Ptr:=LPtr to RPtr do
    write(Opt,Ptr:3,':',Alfabet[Tape[Ptr]]:1);
  writeln(Opt);
end;

procedure BepActie;
{Bepaalt de volgende toestand en het te schrijven symbool}
begin
  if Overgangstabel[Toestand,Gelezen].NieuweToestand<>Dummy then
    GevrActie:=Overgangstabel[Toestand,Gelezen]
  else
    FunctieDef:=false;
end;

begin
{Initialisatie}
  for Toestand:=1 to AantalToestanden do
```

```
  for Symboolnr:=0 to AantalSymbolen do
    Overgangstabel[Toestand,Symboolnr].NieuweToestand:=Dummy;
for Tapekop:=-Tapelimiet to Tapelimiet do
  Tape[Tapekop]:=0;
Ans:=' ';write('Output naar Scherm of Printer (S/P)? ');
Readln(Ans);
assign(Opt,'CON:');
if Ans='P' then
  assign(Opt,'LST:');
rewrite(Opt);
Alfabet[0]:=Blank;Nstap:=0;
writeln(Opt,'Begintoestand is: ',Begintoestand);
writeln(Opt,'Eindtoestanden vanaf: ',EersteHalttoestand);

{Lees de tape in}
assign(TMtape,'B:TMtape.dat');
reset(TMtape);
writeln(Opt);
writeln(Opt,'De inputtape bevat: ');
SymBekend:=0;
Tapekop:=1;
while not eof(TMtape) do
begin
  readln(TMtape,Symbool);
  write(Opt,Tapekop:3,':',Symbool:1);
  Tape[Tapekop]:=BepSymcode(Symbool);
  Tapekop:=Tapekop+1
end;
MeestRechter:=Tapekop;
writeln(Opt);writeln(Opt);

{Lees de vijftupels in}
assign(Tupelfile,'B:Tupels.dat');
reset(Tupelfile);
writeln(Opt,'Het Turingmachineprogramma is:');
while not eof(Tupelfile) do
begin
  read(Tupelfile,Tupel);
  with Tupel do
    writeln(Opt,HuidigeToestand,' ',Leessymbool,' ',Schrijfsymbool,' ',
            Stap,' ',NieuweToestand);
  Toestand:=Tupel.HuidigeToestand;
  Gelezen:=BepSymcode(Tupel.Leessymbool);
  with GevrActie do
  begin
```

```
    SchrijfSymbool:=BepSymcode(Tupel.Schrijfsymbool);
    Stap:=Tupel.Stap;
    NieuweToestand:=Tupel.NieuweToestand;
  end;
  Overgangstabel[Toestand,Gelezen]:=GevrActie
end;writeln(Opt);

{Voer het Turingmachineprogramma uit}
if Spoor then
begin
  writeln(Opt);
  writeln(Opt,'      Nwe Pos Gel Sch Stp');
  writeln(Opt,'  Stp Tst Tpk Sym Sym Rch == Tapeinhoud ==');
end;
Tapekop:=1;MeestLinker:=1;
Toestand:=Begintoestand;Gelezen:=Tape[1];
FunctieDef:=true;TapeVol:=false;Eindtape:=false;
BepActie;
while FunctieDef and not TapeVol do
  with GevrActie do
  begin
    Toestand:=NieuweToestand;
    Tape[Tapekop]:=SchrijfSymbool;
    if Stap='L' then
      begin
          Tapekop:=Tapekop-1;
          if Tapekop<MeestLinker then
            MeestLinker:=Tapekop;
      end
    else if Stap='R' then
      begin
          Tapekop:=Tapekop+1;
          if Tapekop>MeestRechter then
            MeestRechter:=Tapekop;
      end;
    Nstap:=Nstap+1;
    if Spoor and (Alfabet[Gelezen]<>Alfabet[Schrijfsymbool]) then
      ToonResultaat;
    if abs(Tapekop)<=Tapelimiet then
    begin
      Gelezen:=Tape[Tapekop];
      BepActie;
    end
    else
      TapeVol:=true
```

```
    end;
    if TapeVol then
      writeln(Opt,'Tapelimiet bereikt')
    else
      if Toestand<EersteHalttoestand then
        writeln(Opt,'Turingmachine stopt zonder succes')
      else
      begin
        Eindtape:=true;
        ToonResultaat;
      end;
  close(Tupelfile);
end.
^Z
```

Index

ACADEMIC SERVICE INFORMATICA UITGAVEN

AUTOMATISERING EN COMPUTERS

Computers en onze informatiemaatschappij - M.A. Arbib
Computers in de negentiger jaren - G.L. Simons
De informatiemaatschappij - Jan Everink
Op weg naar een risicoloze maatschappij? - Jan Holvast
Basiskennis informatieverwerking - Jan Everink
AIV, Automatisering van de informatieverzorging - Th.J.G. Derksen & H.W. Crins
BIV, Basis van de geautomatiseerde informatieverzorging - Th.J.G. Derksen & H.W. Crins
Organisatie, informatie en computers - David M. Kroenke
Computers in de basisschool - H. Lamers & J.A. Wegkamp
Effectieve toepassingen van computers - M. Peltu

MICROCOMPUTERS

Microcomputers thuis en op school - K.P. Goldberg & D. Sherwood
Bouw zelf een expertsysteem in BASIC - Chris Naylor
Programmeercursus MicrosoftBASIC - Nok van Veen
Werken met bestanden in BASIC - LeRoy Finkel & Jerald R. Brown
Programmeercursus BASIC op de Commodore 64 - Nok van Veen
Werken met bestanden op de Commodore 64 - G. Fisher, L. Finkel & J.R. Brown
Het Electron en BBC Micro boek - Jim McGregor & Alan Watt
Werken met bestanden op de Apple - LeRoy Finkel & Jerald R. Brown
Programmeercursus ApplesoftBASIC - Nok van Veen & Ad van Delft
Programmeercursus MSX BASIC - Nok van Veen
Werken met bestanden in MSX BASIC - LeRoy Finkel & Jerald R. Brown
40 grafische programma's - voor de Commodore 64; voor de Electron en BBC; voor de ZX Spectrum; voor de Apple II, IIe en IIc; in MSX BASIC - Marcel Sutter

MICROPROCESSORS EN ASSEMBLEERTALEN

Procescomputers, basisbegrippen - J.E. Rooda & W.C. Boot
Cursus Z-80 assembleertaal - Roger Hutty
6502 assembleertaal en machinecode voor beginners - A.P. Stephenson
EXAT-handboek - Micro-Teach

BESTURINGSSYSTEMEN

Inleiding besturingssystemen - A.M. Lister
Theorie en praktijk van besturingssystemen - J.L. Peterson & A. Silberschatz
Systeemprogrammatuur en softwareontwikkeling voor microcomputers - E. Verhulst
Bedrijfssystemen - EIT-serie, deel 4
CP/M, het operating system voor microcomputers - J.N. Fernandez & R. Ashley
CP/M 86, een besturingssysteem voor 16 bit microcomputers - J.N. Fernandez & R. Ashley
CP/M voor gevorderden - A. Clarke e.a.
PC DOS, het besturingssysteem van de IBM PC - R. Ashley & J.N. Fernandez
MS DOS, het besturingssysteem voor 16 bit microcomputers - R. Ashley & J.N. Fernandez
UNIX, het standaard operating system - G.J.M. Austen & H.J. Thomassen
De UNIX programmeeromgeving - B.W. Kernighan & R. Pike

PERSONAL COMPUTERS EN PROGRAMMAPAKKETTEN

De IBM PC en zijn toepassingen - Laurence Press
Werken met bestanden in IBM- en GW-BASIC - J.R. Brown & LeRoy Finkel
40 grafische programma's in IBM- en GW-BASIC - Marcel Sutter
Werken met VisiCalc - C. Klitzner & M.J. Plociak Jr.
Multiplan, een hulpmiddel bij de bedrijfsvoering - D.F. Cobb e.a.
Werken met Lotus 1-2-3 - G.T. LeBlond & D.F. Cobb
Lotus 1-2-3: Tips, Trucs en Tegenvallers - D. Andersen & D.F. Cobb
Lotus 1-2-3: Financiële macro's - Thomas W. Carlton
Symphony deel I en II - D.P. Ewing & G.T. LeBlond
dBASE III handboek - George Tsu-der Chou
WordStar stap voor stap - Ruth Ashley & Judi N. Fernandez

PROGRAMMEREN

Een methode van programmeren - Edsger W. Dijkstra & W.H.J. Feijen
Programmeren, met ontwerpen van algoritmen (met Pascal) - J.J. van Amstel
Voortgezet programmeren, het ontwerpen van datastructuren en algoritmen - J.J. van Amstel & J.A.A.M. Poirters

Problemen oplossen met de computer - R.G. Dromey
Inleiding tot het programmeren, deel 1 en 2 - J.J. van Amstel e.a.
Het Groot Pascal Spreukenboek - Henry F. Ledgard e.a.
Software engineering, het bouwen van grote programma's - I. Sommerville
JSP Jackson struktureel programmeren - Henk Jansen
JSP Uitwerkingenboek, JSP Procedureboek - Henk Jansen

PROGRAMMEERTALEN

Aspecten van programmeertalen - J.J. van Amstel & J.A.A.M. Poirters
Programmeertalen, een inleiding - J.J. van Amstel e.a.
Colloquium programmeertalen - red. J.A.A.M. Poirters & G.J. Schoenmaker
BASIC - EIT-serie, deel 3
Cursus BASIC, een practicum handleiding voor BASIC op de PRIME - R. Bloothoofd e.a.
Een programmeercursus in BASIC - Nok van Veen & René Wissing
Cursus Pascal - A. van der Sluis & C.A.C. Görts
Cursus eenvoudig Pascal - A. van der Sluis & C.A.C. Görts
Inleiding programmeren in Pascal - C. van de Wijgaart
Modula-2 - E. Verhulst
Systeemontwikkeling met Ada - Grady Booch
Cursus COBOL - A. Parkin
Cursus FORTRAN 77 - J.N.P. Hume & R.C. Holt
De programmeertaal C - L. Ammeraal
Flitsend Forth - Alan Winfield
Logisch LOGO - Auke Sikma
Programmeren in LISP - L.L. Steels
Micro-PROLOG, programmeren in logica - K.L. Clarke & F.G.McGabe
Inleiding PROLOG - W. Burnham & A. Hall

BESTANDSORGANISATIE, DATABASE EN GEGEVENSANALYSE

Bestandsorganisatie - R.J. Lunbeck & F. Remmen
Database, een inleiding - C.J. Date
Databases, grondslagen voor de logische structuur - F. Remmen
SQL in de praktijk - H.B. Eilers e.a.
Het SQL Leerboek - Rick F. van der Lans
Gegevensanalyse - R.P. Langerhorst

INFORMATIE-ANALYSE EN SYSTEEMONTWERP

Inleiding systeemanalyse en systeemontwerp - W.S. Davis
Systeemontwikkeling zonder zorgen - Paul T. Ward
Systeemontwikkeling volgens SDM - H.B. Eilers
Samenvatting SDM - Pandata
Systeemontwikkeling volgens JSD - Michael Jackson
Informatie-analyse volgens NIAM - J.J.V.R. Wintraecken
Information engineering - J. Blank
Het ontwerpen van interactieve toepassingen en computernetwerken - J.A. Scheltens
EDP Audit - C. de Backer
Management informatiesystemen - G.B. Davis & M.H. Olson
Prototyping, een instrument voor systeemontwerpers - red. T. Hoenderkamp & H.G. Sol
Het ontwikkelen van informatiesystemen met prototyping - R. Vonk
Simulatie, een moderne methode van onderzoek - S.K.T. Boersma & T. Hoenderkamp

EXPERTSYSTEMEN EN KUNSTMATIGE INTELLIGENTIE

Kunstmatige intelligentie - Patrick H. Winston
Expertsystemen - Henk de Swaan Arons & Peter van Lith
Ontwikkelingen in expertsystemen - red. A. Nijholt & L.L. Steels

THEORETISCHE INFORMATICA EN SYSTEEMPROGRAMMATUUR

Systeemprogrammatuur - H. Alblas
Vertalerbouw - H. Alblas e.a.

OPERATIONELE RESEARCH

Operationele research - Y.M.I. Dirickx e.a.
Lineaire programmering als hulpmiddel bij de besluitvorming - S.W. Douma

Problemen oplossen met de computer - R.G. Dromey
Inleiding tot het programmeren, deel 1 en 2 - J.J. van Amstel e.a.
Het groot Pascal Sprookjesboek - Henry & Ledgard e.a.
Software engineering, het bouwen van grote programma's - I. Sommerville
ISP Junior: structureel programmeren - Henk Jansen
ISP Uitbreidingenboek, ISP Procedureboek - Henk Jansen

PROGRAMMEERTALEN

Aspecten van programmeertalen - J.J. van Amstel & T.A.A.M. Poirters
Programmeertalen, een inleiding - J.J. van Amstel e.a.
Colloquium programmeertalen - red. T.A.A.M. Poirters & G.J. Schoenmakers
BASIC + ELT extra, deel
Cursus BASIC, een praktische handleiding voor BASIC op de PRIME - N. Bloemendaal
Een programmeercursus in BASIC - Bok van Vuren & Henk Mesure
Cursus Pascal - A. van der Sluis & C.A.G. Gerts
Cursus gestructureerd Pascal - A. van der Sluis & C.A.G. Gerts
Inleiding programmeren in Pascal - G. van de Wijgaart
Modula-2 - E. Verhulst
Systeemontwikkeling met Ada - Grady Booch
Cursus COBOL - A. Parkin
Cursus FORTRAN 77 - J. P. Hume & R.C. Holt
De programmeertaal C - L. Ammeraal
Cursus Forth - Alan Winfield
Cursus LOGO - André Bisset
Programmeren in LISP - L. L. Steele
Micro-PROLOG: programmeren in logica - K.L. Clarke & F.G. McCabe
Cursus PROLOG - W.F. Clocksin & A. Hall

BESTANDSORGANISATIE, DATABASE EN GEGEVENSANALYSE

Bestandsorganisatie - B.J. Lumbeek & F. Blokker
Databases, een inleiding - C.J. Date
Databases, praktisch over de logische structuur - F. Remmen
SQL in de praktijk - H.B. Eilers e.a.
Het SQL leerboek - Rick F. van der Lans
Gegevensanalyse - H. Z. Langerhorst

INFORMATIE ANALYSE EN SYSTEEMONTWERP

Inleiding systeemanalyse en systeemontwerp - W.S. Davis
Systeemontwikkeling zonder zorgen - Paul T. Ward
Systeemontwikkeling volgens SDM - H.B. Eilers
Samenvatting SDM - Pandata
Systeemontwikkeling volgens JSD - Michael Jackson
Informatieanalyse volgens NIAM - J.J.V.R. Wintraecken
Information engineering - J. L. Blank
Het ontwerpen van interactieve toepassingen en computerinterfaces - J.A. Schelfers
EDP Audit - E.J. de Ruijter
Management informatiesystemen - G.B. Davis & M.H. Olson
Prototyping, een instrument voor systeemontwerpers - red. H. Boomkamp & R.G. [illegible]
Het ontwikkelen van informatiesystemen met prototyping - R. Vonk
Simulatie: een moderne methode van onderzoek - S.K.T. Boersma & T. Hoogerkamp

EXPERTSYSTEMEN EN KUNSTMATIGE INTELLIGENTIE

Kunstmatige intelligentie - Patrick H. Winston
Expertsystemen - Henk de Swaan Arons & Peter van Lith
Ontwikkelingen in expertsystemen - red. A. Nijholt & L.K. Steels

THEORETISCHE INFORMATICA EN SYSTEEMPROGRAMMATUUR

Systeemprogrammatuur - H. Alblas
Vertalerbouw - H. Alblas e.a.

OPERATIONELE RESEARCH

Operationele research - L.M.J. Ulrickx e.a.
Lineaire programmering als hulpmiddel bij de besluitvorming - S.W. Douma

INFORMATIE OVER DEZE PUBLIKATIES BIJ:

Academic Service, Postbus 96, 2870 AB Schoonhoven, tel. 01823-6[illegible]

Het ontwerpen, analyseren en efficiënt uitwerken van algoritmen is een belangrijke activiteit binnen de informatica. Dit boek tracht de meest fundamentele vragen te beantwoorden die men zich hierbij kan stellen: 'Waar liggen de grenzen van de mogelijkheden van computers?', 'Bestaat er een algoritme voor het oplossen van een gegeven probleem?' en zo ja 'Bestaat er ook een efficiënt werkend algoritme?'.
De berekenbaarheidstheorie biedt de mogelijkheid problemen te classificeren in oplosbare en onoplosbare problemen en maakt daarbij gebruik van het Turingmachinemodel. De aan het slot van dit boek behandelde complexiteitstheorie geeft ons vervolgens antwoord op de vraag hoe 'moeilijk' een bepaald probleem is en verdeelt daartoe oplosbare problemen verder onder in handelbare en onhandelbare problemen. Het grootste deel van deze theorie is van recente datum (na 1970) en de theorie is nog volop in ontwikkeling. Kennisnemen van de huidige stand van zaken is daarom voor iedere informaticastudent en voor ieder die in informatica geïnteresseerd is een 'must'!

> ACADEMIC SERVICE

ISBN 90 6233 243 9
NUGI 852/112